CS 428/528 Project 3: Overlay Network

Part 1 and 2 due 4/22 9pm; rest due May 3, 9pm

Introduction: This project asks you to implement an application level network (what is known as an overlay network) using processes and socket connections. I would like you to work in pairs, but individual efforts are also acceptable. We will have a help session to get you started. As much as possible, you are expected to share the load equally with your partner. In addition, each partner is responsible for understanding the whole implementation. This is something that we will look for explicitly during grading and its possible to give unequal grades to different individuals within the group.

Each node in your network is a process and the links are emulated over a UDP socket (similar to your project 2 server). In this project, you will first create this virtual infrastructure and implement routing for it. In the second part, you will implement forwarding and a simplified interface for it. You can now test your implementation by sending a packet from one node to another and watch it get forwarded through the intermediate hops.

Your overlay layer does not have to do any frame translation or fragmentation. However, it needs to figure out forwarding and routing. You will test how your routing can keep track of a dynamically changing topology.

1 Part I: Creating your virtual Infrastructure

Each "host" will be a process running on a known machine/port. Note that you cannot just use the same "hardcoded" port for all your hosts if you want to be able to run some of them on the same machine. A configuration file will specify the topology of the network (described below).

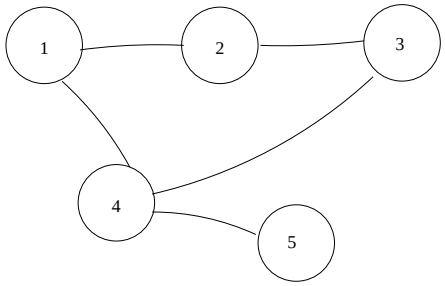

Figure 1: Example Network

Configuration File: The configuration file describes the network topology. It should have a line for every host in the network containing, the node number, the host where it will run, the port numbers, and the nodes it is connected to. To simplify things, I suggest using two different ports, one for data packets and one for routing/hello packets. For example, for the sample network shown in Figure 1, assume nodes 1 will be "simulated" on sunblade10.cc.binghamton.edu ports 5001, 5002 for data and routing sockets, while nodes 4 will be simulated on sunblade11.cc.binghamton.edu ports 5001 and 5002. You have to start the processes manually at the machine names/ports in the configuration file. We are abstracting away the link layer, so do not worry about ARP or a hardware address. An example configuration file would look something like:

```
//Format is:
//Node# machine_name           data_port routing_port node1 node2 ...
   1    class00.cs.binghamton.edu 5000   5001   2 4
   2    class01.cs.binghamton.edu 5002   5003   1 3
   3    class01.cs.binghamton.edu 5000   5001   2 4
   4    class00.cs.binghamton.edu 5002   5003   1 3 5
   5    class00.cs.binghamton.edu 5004   5005   4
```

The first line says that the process for node 1 is running on class00, with data port 5000, routing port 5001 and connects directly to nodes 2 and 4. The file is for purposes of starting up the network and determining neighbors only. Your hosts cannot use any information about nodes other than its direct connections. So, node 1 may not learn from this file the location of nodes 3 and 5 since it does not connect directly to them. Since each of your host processes reads this file, they know what machines they are connected to: each node connects to the nodes listed at the end after the port numbers. The host knows their port number from their line in the configuration file.

The reason we separate the data port from the routing port is to simplify the interaction between the routing protocol and the forwarding of the data packets. If both types of packet were sent to the same port then you have to separate packets based on their type and to manage shared buffers between the forwarding thread and the routing thread.

Also, for simplicity of the implementation, we elected to have one port to receive all hello/routing packets on for each "host". This means that packets sent on different physical links to the one host will both appear on this port. When you receive a packet, sometimes you need to determine who the previous hop to this packet was (for example for hello messages . There are two ways to do this:

1. have a previous hop field in the packet header so you can tell who this packet came from and match it against your neighbor list to know who it was received from; or

2. check the IP address and port number that are returned by the recvfrom() call.

I prefer that you use the first one so that the test client described in Part IV below. This allows the test client to "spoof" packets from other hosts to generate commands like create-link and remove-link (also described below)

2 Part II: Routing and Hello

To implement routing, you need two protocols. A hello protocol that keeps neighbors aware of each other, and a link-state or distance vector protocol to implement routing (if you prefer to do something else like source routing that is fine too as long as you dynamically construct the forwarding information). If you use distance vector, then your routing messages can serve as your hello messages as well (when you receive an update from a neighbor, that serves as a hello message as well).

Both protocols should be triggered periodically based on a timer. Also, the information that you learn from the these messages should be timed out after an appropriate time to live (TTL). For both of these things you need a timer that periodically wakes a separate routing thread up, which: (1) sends routing and hello messages when it is time; (2) reduces the TTL on existing information, deleting ones whose TTL reaches 0; and (3) receives routing messages, updates routing information and constructs forwarding table (for example, by solving Dijkstra's shortest path algorithm for link state).

Note that when you receive new information, or a new message repeating information you know, you set the TTL on this information to the maximum TTL value (which you should select to balance reaction to dynamic changes vs. overhead as we discussed in class).

In our implementation, the only data structure that the threads share is the routing table – the routing thread computes/updates it and the forwarding thread uses it to find out the next hop. You have to use locks or semaphores to order access to this table (more later).

One potential issue in this kind of simulation is deadlock that arises due to recvfrom() being called on a socket that does not have data on it. To address this problem, use the select() or poll() call to make sure that there is data on the socket before you. Also, try to send messages before you receive on your incoming sockets.

Once you receive routing messages, you have to update your forwarding table. Recall that this table maps a destination address to a next hop address. How you update this table depends on your choice of routing protocol (distance vector vs. link state). You can use the number of hops as your cost metric (i.e., all links have identical cost of 1).

Please make sure to lock the forwarding table (Which is shared by the forwarding thread described below) before you update it. You may want to consider having two tables: a production table which is used for the routing,

and a staging table where you do updates. Then, you can switch the staging and production tables by updating a pointer minimizing the interaction between the threads (i.e., the size of the mutex region). This is optional.

Depending on your choice of routing protocol, you need to design your hello and routing packet formats.

You should have a debug option that, when set, prints detailed debug information about the received/sent packets and the state of the routing table whenever it changes.

3 Part III: Forwarding and Network Layer Interface/Packet Format

Each host will receive packets that arrive from its neighbors on the data socket and forward them to the next hop (unless they are destined to this node itself). This will be implemented by the forwarding thread whose main responsibility is to receive data from neighboring links and forward them on if they are not destined to the local host.

For simplicity and generality of the infrastructure design, we elected to have one port on each host that will receive the data packets.

First, we have to define our data packet format. If you like, you are free to design your own, or you can use this format:

```
[16-bit Packet Id     ]
[16-bit Protocol Id   ]
[16-bit Source Id     ]
[16-bit Destination Id]
[16-bit Packet Length ]
[16-bit previous hop  ]
[16-bit TTL           ]
[ Data       ...      ]
[   ....              ]
```

Packet Id can be assigned sequentially by each host on each link. Protocol Id can be used as demux key so that when the packet is received at the final destination, you know what the next layer is. This allows you to build multiple different protocls on top of this project. Source Id and Destination Id are the Node Id's for the originating and destination packet respectively. Packet length specifies the length of this packet in bytes. Previous hop specifies the node Id of the immediate node that sent this packet ("link-layer" source address). You have to set this field to your own id in packets that you forward. TTL is the time to live on the packet that should be decremented each time the packet is forwarded. Try to limit the data size to around 1KByte.

Your forwarding thread is responsible for the following. When you send a packet from one "host" to another "host" you first look in your forwarding table and send the packet to the next hop link (for example, as was given to you in the configuration file). At every intermediate node, when you receive a packet, if it is not destined to you, you have to forward it over to the next hop as per your forwarding table. One exception is if the TTL on the packet is 1 when you receive it, at that point you have to drop the packet. Before you forward the packet, decrement the TTL field and the previous hop field. You should print debug information (the packet header) whenever you generate a new packet, receive a packet, drop a packet or forward a packet. You have to acquire a lock before checking the forwarding table in case it is being updated by the routing thread.

4 Part IV: Client to control testing the implementation

You have to show that your routing protocol works and your forwarding works. The first test is to show that a packet sent from any node to any other node (that it is not necessarily directly connected to) will arrive at the destination after being forwarded by the intermediate nodes. This will also show that basic routing works. However, I would also like to see how your routing protocol adapts to changes in the topology like a link being removed or a new link being added.

We need a way to be able to give commands like the generation of a new packet at a certain host, and the addition/removal of links. One way to do that is to allow one of our sockets to handle special command packets. For example, the hello/routing socket could be made to accept additional commands as follows:

(1) generate packet to destination n

(2) tear down this connection: the effect of this is to remove the link from the neighbor list

(3) add new-link: add a new link to the neighbor list

Your "hello"/"routing" protocol should have an opcode or type field in the packet header to allow it to figure out whether this is a hello or routing packet or one of the commands above.

Write a client to accept command line arguments to do the three commands above. This client should read the configuration file and send command packets directly to the appropriate ports. So, if the command is "`generate_packet` 1 2", it will send a `generate_packet` command to the hello socket of 1 to create a packet destined to 2. When 1 receives this message, it will create a data packet and send it along to 2 (possibly through intermediate neighbors). For new-link and tear-down commands, it should generate the tear-down or add-new-link command packet and send it to the appropriate neighbors to tear down a link or establish a new one.

Once you change the topology, you should show that your routing protocol reacts and your packets follow different paths. Experiment with different topologies and try to figure out the characteristics of your routing protocol (how fast it adapts, how fast it converges, do you get loops?).

5 Part V: Functionality on top of Overlay

Please note that there are 5 subsections to this part; you have to do **two** of them only as follows. If you implement all five, you can skip the final exam. If you take this option, you can have additional time to finish.

Some of the functionalities we ask you to implement are intentionally under specified to give you space to be creative and to design. We will favor cleaner more realistic designs to hacks that just illustrate that something works.

5.1 Distributed Hash Tables/P2P network

Implement distributed hash tables on top of your overlay. Essentially, you should create a hash function that hashes any string and maps it to a node in the overlay. Now, documents can be hashed to different peers in your overlay, and anyone that is looking for a document should be able to find the document at the appropriate node if it is there.

Your implementation should support two operations: **put** and **get**. Put places a document provided by name at a node while get queries the node and returns whether the document was found or not.

5.2 Ping and Traceroute

Implement ping and traceroute. This requires you to implement a simple ICMP protocol on top of the basic network layer. When a packet gets droped due to TTL, you generate an "ICMP" packet back to the original source. This can then be used to build traceroute as we discussed in class.

To support ping, you have to send an ICMP ping packet which, when received at the destination, sends a reply packet back to the source. You can use the standard unix ping and traceroute utilities to get an idea about what information to display.

5.3 Supporting Multicast

In this part you will support multicast in your overlay. With multicast, we will have one or more multicast groups, each with several members. When you send to the group as destination, all the members in the group should receive the packet. You can implement any multicast algorithm that builds a spanning tree (i.e., anything but the brute force methods such as unicast to all group members or broadcast to everyone). The easiest would probably be Core based Tree which is equivalent to sparse mode PIM.

In the configuration file, add the groups which a node is member of at the end of its line. These groups could have id of 1000 or more so that you do not mistake them for regular connections. For example, the line for node 1 may become:

1 bingsuns.binghamton.edu 7000 7001 3 4 1001 1002

to say that its connected to 3 and 4 and a member of groups 1001 and 1002. You may assume that the RP (or the tree root) for a group is known (for example, by taking some hash of the group number such as groupNumber mod totalNodes).

You should have a multicast forwarding tree that includes an entry for every multicast group. Recall that an intermediate node may have to forward a multicast packet to multiple outgoing links; the multicast forwarding table entry should have for each group we are a member of the set of links it should forward the packet on. That is, there are multiple next hops possibly for each multicast entry. So, your data structure should have a way to include multiple hops (a linked list, or a fixed number array of next hops).

Each next hop should have its own TTL entry. The TTL is set whenever a join request comes in on that link. It is decremented with every heartbeat.

Joining and Leaving: Periodically, each node should send a join unicast message to the RP for each of its groups. As the join packet is forwarded towards the RP, at each intermediate node it is processed this way:

1- If node is already part of the tree: when we receive a packet for a group we are already part of the tree for, we add the receiving link to forwarding table entry for this multicast group. Also, we generate an ACK back to the source of the join to indicate that the join was successful.

2- If node is not part of the tree, it adds itself to it by creating a forwarding table entry for the group, with one outgoing link (the link it received the packet from). The node should forward the join packet towards the RP.

Leaving is implemented by stopping the send of the Join message. When TTL expires at intermediate nodes, the node will be removed from the multicast tree automatically. So, really you dont have to do anything for leaving

Forwarding multicast packets: Assume that only multicast members can send a packet to the group. When a node sends a packet to the group, the packet should be flooded in the tree. This is done by sending the packet out on all the next hops for the multicast tree other than the one that the packet is received on. So, when you receive a multicast packet from neighbor X, walk the next hop list for this multicast group and send the packet out on every hop except the one leading back to X.

5.4 Implement a Reliable end-to-end protocol

In this part of the assignment you have to implement a *reliable* end to end protocol. What you implemented so far is the equivalent of IP. Recall that an end-to-end protocol provides a process to process connectivity. This is done by demultiplexing from network protocol (what you implemented last project) to the appropriate protocol using a protocol number field in the header (dont forget to remove the network header). Once your end to end protocol receives the packet, it should demux to the appropriate “socket” using a port number field.To build an end to end protocol on top of it, you have to first include a transport protocol field in packet header and demux to the appropriate protocol. For each protocol you support, have a limited size circular buffer that you place packets in. You have to implement a producer consumer circular buffer object (or data structure) that is safely protected using locks. A circular buffer is simply a fixed size buffer. Most Operating Systems textbooks have examples of how to implement that.

When a packet is received by the forwarding thread that is destined for this node, instead of discarding it as we did before, we should now decapsulate it and pass it to the end to end protocol. This is done by taking the data part of the packet (i.e., removing the header for the network protocol) and placing it at the end of the buffer for the protocol (which specified as the demux key in the network protocol header). If the buffer is full, throw the packet away. If the buffer does not exist, this means that the protocol is not supported and you can just drop the packet (this case should not really happen here).

You can implement a transport protocol as a separate thread (Its enough to implement a single transport protocol.) The thread reads packets from its buffer and processes them. The data part of the original packet which is placed by the forwarding thread in the protocol buffer includes the header for your transport protocol. At a minimum this should include the information in the UDP header (check the book for that). This includes source and destination node id and port numbers. Use the destination port number to demux to the appropriate socket buffer. Again, this is a circular

buffer that is written to by the transport protocol and read by the application thread that owns this socket.

To implement reliability, you should have some form of sliding window algorithm running. The flavor of the sliding window algorithm you want to implement is up to you. You have to generate acknowledgements for correctly received packets. You have to figure out how to implement timers and retransmissions. Stop-and-wait at a minimum is expected. You can do Go-back-N or Selective Repeat for extra credit. For timers, you can use software timers – that is, set the retransmit timer to be some multiple of the heartbeats of the transport thread. Every time it wakes up it decrements the timer. If it becomes 0, retransmission is initiated. If an ACK is received, the packet is freed and the retransmit timer cancelled.

You should create one or more application threads that then send packets to each other. You can just hard code the ports they are running on for now, but you should clearly show that you can run multiple connections from the same host and not get them mixed up.

5.5 Tunneling

Implement tunneling (IP-in-IP) either as a separate protocol or within IP as an option (i.e., when the packet reaches the destination, it checks if its an IP-in-IP packet and processes it by stripping the outer packet and forwarding on.

Illustrate that your implementation works by building an application (VPN, Mobile IP, onion routing) that uses your tunneling implementation.

This part is intentionally under specified to give you the space to make assumptions and design.

Forming and Unforming Packets

UDP deals with character buffers in its sendto and recvfrom calls. That is, your packet should be packaged in a buffer before you send it over. This means that you have to assemble packets yourself into a character buffer – you cannot just pass a data structure to send/receive.

To accomplish this, use the call memcpy whose prototype looks like this:

```
void *memcpy(void *dest, const void *src, size_t n);
```

which copies n bytes from memory location src to memory location dest.

Here is an example of how to create a packet that has a length field (4 bytes), followed by a type field of length 4 bytes, followed by a data field of length 256 bytes:

```
char buffer[1024]; //I'm gonna place the packet here
int type, length; //integers that hold the type and the length
char payload[256]; //data initialized to something

memset (buffer, 0, 1024); //zero out buffer

memcpy ((void *)buffer,(void *) &length, 4); //copy 4 bytes from
                                             // length to buffer

memcpy ((void *)(buffer+4),(void *) &type, 4); //copy 4 bytes from
                                               //type to address (buffer+4)
       //i.e., just after length
memcpy ((void *)(buffer+8),(void *)payload, 256); //copy 256 bytes from
                                                  //payload to buffer+8
```

On the receiver side, once you receive the packet, it will be placed in a character buffer. You can unpack it by doing a copy from the buffer to the variables (e.g., by switching the first two arguments in the memcpy calls above).