CSE 533 (Fall 2015) - Assignment 4

Due date: Sunday, Dec. 6, at 11:55pm

You are to work on this assignment in groups of two students.

I. Overview

This assignment consists of roughly three tasks:

- you will develop an application that uses raw *IP* sockets to 'walk' around an ordered list of nodes (given as a command line argument at the '*source*' node, which is the node at which the tour was initiated), in a manner similar to the *IP SSRR* (Strict Source and Record Route) option.
- At each node, the application *ping*s the preceding node in the tour. However, unlike the *ping* code in *Stevens*, you will be sending the *ping ICMP echo request* messages through a *SOCK_RAW*-type *PF_PACKET* socket and implementing *ARP* functionality to find the *Ethernet* address of the target node.
- Finally, when the 'walk' is completed, the group of nodes visited on the tour will exchange multicast messages.

Your code will consist of two process modules, a '*Tour*' application module (which will implement all the functionality outlined above, except for *ARP* activity) and an *ARP* module.

The following should prove to be useful reference material for the assignment:

- Sections 21.2, 21.3, 21.6 and 21.10, Chapter 21, on Multicasting.
- Sections 27.1 to 27.3, Chapter 27, on the *IP SSRR* option.
- Sections 28.1 to 28.5, Chapter 28, on raw sockets, the *IP_HDRINCL* socket option, and *ping*.
- Sections 15.5, Chapter 15, on *Unix* domain *SOCK_STREAM* sockets.
- Figure 29.14, *p*. 807, and the corresponding explanation on *p*. 806, on filling in an *IP* header when the *IP_HDRINCL* socket option is in effect.
- The Lecture Slides on *ARP & RARP* (especially Section 4.4, *ARP Packet Format*, and the Figure 4.3 it includes).
- The link http://www.pdbuchan.com/rawsock/rawsock.html contains useful code samples that use IP raw sockets and PF_PACKET sockets. Note, in partcular, the code "icmp4_ll.c" in Table 2 for building an echo request sent through a PF_PACKET SOCK_RAW socket.

II. The *VMware* environment

You will be using the same *vm1* , ... , *vm10* nodes you used for Assignment 3. However, unlike Assignment 3, you should use only interfaces *eth0* and their associated *IP* addresses and ignore the other *Ethernet* interfaces that nodes have (interfaces *eth0* make *vm1* , ... , *vm10* look as if they belong to the same *Ethernet LAN* segment *IP* network 130.245.156.0/24). Note that, apart from the primary *IP* addresses associated with interfaces *eth0*, some nodes might also have one or more alias *IP* addresses associated with their interface *eth0*.

III. Tour application module specifications

1. The application will create a total of four sockets: two *IP* raw sockets, a *PF_PACKET* socket and a *UDP* socket for multicasting.

    - We shall call the two *IP* raw sockets the '*rt* ' ('route traversal') and '*pg* ' ('ping') sockets, respectively. The *rt* socket should have the *IP_HDRINCL* option set. You will only be receiving *ICMP echo reply* messages through the *pg* socket (and not sending *echo request*s), so it does not matter whether it has the *IP_HDRINCL* option set or not.

      The *pg* socket should have *protocol* value (*i.e.*, protocol demultiplexing key in the *IP* header) *IPPROTO_ICMP*.

      The *rt* socket should have a *protocol* value that identifies the application - *i.e.*, some value other than the *IPPROTO_XXXX* values in */usr/include/netinet/in.h*. However, remember that you will all be running your code using the same *root* account on the *vm1 , ... , vm10* nodes. So if two of you happen to choose the same *protocol* value and happen to be running on the same *vm* node at the same time, your applications will receive each other's *IP* packets. For that reason, try to choose a *protocol* value for your *rt* socket that is likely to be unique to yourself.

    - The *PF_PACKET* socket should be of type *SOCK_RAW* (**not** *SOCK_DGRAM*). This socket should have a *protocol* value of *ETH_P_IP* = 0x0800 (*IPv4*).

    - The *UDP* socket for multicasting will be discussed below. **Note that, depending on how you choose to bind that socket, you might need to have <u>two</u> *UDP* sockets** for multicast communication – see bottom of *p*. 576, Section 21.10.

2. Your application will have to be running on every *vm* node that is included in the tour.

    - When evoking the application on the source node, the user supplies a sequence of *vm* node names (**not** *IP* addresses) to be visited in order. This command line sequence starts with the next node to be visited from the *source* node (*i.e.*, it does not start with the *source* node itself). The sequence can include any number of repeated visits to the same node. For example, suppose that the source node is *vm3* and the executable is called *tour_jsmith* :

      [root@vm3/root]# tour_jsmith *vm2 vm10 vm4 vm7 vm5 vm2 vm6 vm2 vm9 vm4 vm7 vm2 vm6 vm5 vm1 vm10 vm8*

      (but note that the tour does not necessarily have to visit **every** *vm* node; and the same node should not appear consequentively in the tour list – *i.e.*, the next node on the tour cannot be the current node itself).

    - The application turns the sequence into a list of *IP* addresses for source routing. It also adds the *IP* address of the *source* node itself to the beginning of the list. The list thus produced will be carried as the payload of an *IP* packet, **not** as a *SSRR* option in the packet header. It is our application which will ensure that every node in the sequence is visited in order, not the *IP SSRR* capability.

    - The source node should also add to the list an *IP* multicast address and a port number of its choice. It should also join the multicast group at that address and port number on its *UDP* socket. The TTL for outgoing multicasts should be set to 1.

    - The application then fills in the header of an *IP* packet, designating itself as the *IP* source, and the next node to be visited as the *IP* destination. The packet is sent out on the *rt* socket. Note that on *Linux*, all the fields of the packet header must be in network byte order (Stevens, Section 28.3, *p*. 737, the fourth bullet point).

    - When filling in the packet header, you should explicitly fill in the *identification* field (recall that, with the *IP_HDRINCL* socket option, if the *identification* field is given value 0, then the kernel will set its value). Try to make sure that the value you choose is likely to be unique to yourself (for reasons similar to those explained with respect to the *IPPROTO_XXXX* in 1. above).

3. When a node receives an *IP* packet on its *rt* socket, it should first check that the *identification* field carries the right value (this implies that you will hard code your choice of *identification* field value determined in item 2 above in your code). If the *identification* field value does not check out, the packet is ignored. For a valid packet :

   - **Print out a message along the lines of:**

     <*time*> received source routing packet from <*hostname*>

     <*time*> is the current time in human-readable format (see lines 19 & 20 in Figure 1.9, *p*. 14, and the corresponding explanation on *p*. 14*f*.), and <*hostname*> is the host name corresponding to the source *IP* address in the header of the received packet.

   - If this is the first time the node is visited, the application should use the multicast address and port number in the packet received to join the multicast group on its *UDP* socket. The TTL for outgoing multicasts should be set to 1.

   - The application updates the list in the payload, so that the next node in the tour can easily identify what the next hop from itself will be when it receives the packet. How you do this I leave up to you. You could, for example, include as part of the payload a pointer field into the list of nodes to be visited. This pointer would then be updated to the next entry in the list as the packet progresses hop by hop (see Figure 27.1 and the associated explanation on *pp*. 711-712). Other solutions are, of course, possible. The application then fills in a new *IP* header, designating itself as the *IP* source, and the next node to be visited as the *IP* destination. The *identification* field should be set to the same value as in the received packet. The packet is sent out on the *rt* socket.

   - The node should also initiate *ping*ing to the preceding node in the tour (the *IP* address of which it should pick up from the header of the received packet). However, unlike the Stevens *ping* code, it will be using the *SOCK_RAW*-type *PF_PACKET* socket of item 1 above to send the *ICMP echo request* messages.

     Before it can send *echo request* messages, the application has to call on the *ARP* module you will implement to get the *Ethernet* address of this preceding / 'target' node; this call is made using the *API* function *areq* which you will also implement (see sections *ARP module specifications* & *API specifications* below). **Note that *ARP* has to be evoked <u>every</u> time the application wants to send out an *echo request* message, and not just the first time.**

     An *echo request* message has to be encapsulated in a properly-formulated *IP* packet, which is in turn encapsulated in a properly-formulated *Ethernet* frame transmitted out through the *PF_PACKET* socket ; otherwise, *ICMP* at the *target* node will not receive it. You will have to modify Stevens' *ping* code accordingly, specifically, the *send_v4* function. In particular, the *Ethernet* frame must have a value of *ETH_P_IP* = 0x0800 (*IPv4* – see <*linux/if_ether.h*>) in the *frame type* / '*length*' field ; and the encapsulated *IP* packet must have a value of *IPPROTO_ICMP* = 0x01 (*ICMPv4* – see <*netinet_in.h*>) in its *protocol* field.

     You should also simplify the *ping* code in its entirety by stripping all the 'indirection' *IPv4* / *IPv6* dual-operability paraphernalia and making the code work just for *IPv4*. Also note that the functions *host_serv* and *freeaddrinfo*, together with the associated structure *addrinfo* (see Sections 11.6, 11.8 & 11.11), in Figures 27.3, 27.6 & 28.5 (*pp*. 713, 716 & 744*f*., respectively) can be replaced by the function *gethostbyname* and associated structure *hostent* (see Section 11.3) where needed. Also, there is no '-v' verbose option, so this too should be stripped from Stevens' code.

     When a node is ready to start *ping*ing, **it first prints out a 'PING' message similar to lines 32-33 of Figure 28.5, *p*. 744**. It then builds up *ICMP echo request* messages and sends them to the *target* node every 1 second through the *PF_PACKET* socket. It also reads incoming *echo response* messages off the *pg* socket, in response to which **it prints out the same kind of output as the code of Figure 28.8, *p*. 748**.

     If this node and its preceding node have been previously visited in that order during the tour, then *ping*ing would have already been initiated from the one to the other in response to the first visit, and

nothing further should nor need be done during second and subsequent visits.

In light of the above, note that once a node initiates *ping*ing, it needs to read from both its *rt* and *pg* sockets, necessitating the use of the *select* function. As will be clear from what follows below, the application will anyway be needing also to simultaneously monitor its *UDP* socket for incoming multicast datagrams.

4. When the last node on the tour is reached, and if this is the first time it is visited, it joins the multicast group and starts *ping*ing the preceding node (if it is not already doing so). After a few *echo replies* are received (five, say), it sends out the multicast message below on its *UDP* socket (*i.e.*, the node should wait about five seconds before sending the multicast message) :

<<<<< This is node *vm*$_i$. Tour has ended . Group members please identify yourselves. >>>>>

where *vm*$_i$ is the name (**not** *IP* address) of the node. **The node should also print this message out on *stdout* preceded, on the same line, by the phrase:**

Node vm*$_i$. *Sending: <then print out the message sent>.

a. **Each node** *vm*$_j$ receiving this message should **print out the message received preceded, on the same line, by the phrase:**

Node vm*$_j$. *Received <then print out the message received>.

b. **Each such node in step a above should then immediately stop its *ping*ing activity.**

c. The node should then send out the following multicast message:

<<<<< Node *vm*$_j$. I am a member of the group. >>>>>

and print out this message preceded, on the same line, by the phrase:

Node vm*$_j$. *Sending: <then print out the message sent>.

d. **Each node receiving these second multicast messages** (*i.e.*, the messages that nodes – including itself – sent out in step c above) **should print each such message out preceded, on the same line, by the phrase:**

Node vm*$_k$. *Received: <then print out the message received>.

e. Reading from the socket in step d above should be implemented with a 5-second timeout. When the timeout expires, the node should print out another message to the effect that it is terminating the *Tour* application, and gracefully exit its *Tour* process.

f. Note that under Multicast specifications, the last node in the tour, which sends out the *End of Tour* message, should itself receive a copy of that message and, when it does, it should behave exactly as do the other nodes in steps a. – e. above.

IV. ARP module specifications

Your executable is evoked with no command line arguments. Like the *Tour* module, it will be running on every *vm* node.

1. It uses the *get_hw_addrs* function of Assignment 3 to explore its node's interfaces and build a set of <*IP* address , *HW* address> matching pairs for all ***eth0*** interface *IP* addresses (including alias *IP* addresses, if any).

Write out to *stdout* in some appropriately clear format the address pairs found.

2. The module creates two sockets: a *PF_PACKET* socket and a *Unix* domain socket.

    - The *PF_PACKET* should be of type *SOCK_RAW* (**not** type *SOCK_DGRAM*) with a *protocol* value of your choice (but **not** one of the standard values defined in <*linux/if_ether.h*>) which is, hopefully, unique to yourself. This value effectively becomes the *protocol* value for your implementation of *ARP*. Because this *protocol* value will be carried in the *frame type* / '*length*' field of the *Ethernet* frame header (see Figure 4.3 of the *ARP* & *RARP* handout), **the value chosen should be not less than 1536 (0x600)** so that it is not misinterpreted as the length of an *Ethernet 802.3* frame.

    - The *Unix* domain socket should be of type *SOCK_STREAM* (**not** *SOCK_DGRAM*). It is a **listening** socket bound to a 'well-known' *sun_path* file. This socket will be used to communicate with the function *areq* that is implemented in the *Tour* module (see the section *API specifications* below). In this context, *areq* will act as the client and the *ARP* module as the server.

3. The *ARP* module then sits in an infinite loop, monitoring these two sockets.

4. As *ARP* request messages arrive on the *PF_PACKET* socket, the module processes them, and responds with *ARP* reply messages as appropriate.

    The protocol builds a 'cache' of matching <*IP* address , *HW* address> pairs from the replies (and requests – see below) it receives. For simplicity, and unlike the real *ARP*, we shall not implement timing out mechanisms for these cache entries.

    A cache entry has five parts: (i) *IP* address ; (ii) *HW* address ; (iii) *sll_ifindex* (the interface to be used for reaching the matching pair <(i) , (ii)>) ; (iv) *sll_hatype* ; and (v) a *Unix*-domain connection-socket descriptor for a connected client (see the section *API specifications* below for the latter three). When an *ARP* reply is being entered in the cache, the *ARP* module uses the socket descriptor in (v) to send a reply to the client, closes the connection socket, and deletes the socket descriptor from the cache entry.

    Note that, like the real *ARP*, when an *ARP* request is received by a node, and if the request pertains to that receiving node, the *sender*'s (see Figure 4.3 of the *ARP* & *RARP* handout) <*IP* address, *HW* address> matching pair should be entered into the cache if it is not already there (together, of course, with (iii) *sll_ifindex* & (iv) *sll_hatype*), or updated if need be if such an entry already exists in the cache.

    If the *ARP* request received does **not** pertain to the node receiving it, but there is already an entry in that receiving node's cache for the *sender*'s <*IP* address, *HW* address> matching pair, that entry should be checked and updated if need be. If there is no such entry, no action is taken (in particular, and unlike the case above, **no new entry should be made in the receiving node's cache of the *sender*'s <*IP* address, *HW* address> matching pair if such an entry does not already exist**).

5. *ARP* request and reply messages have the same format as Figure 4.3 of the *ARP* & *RARP* handout, but with an extra 2-byte identification field added at the beginning which you fill with a value chosen so that it has a high probability of being unique to yourself. This value is to be echoed in the reply message, and helps to act as a further filter in case some other student happens to have fortuitously chosen the same value as yourself for the *protocol* parameter of the *ARP PF_PACKET*. Values in the fields of our *ARP* messages must be in network byte order. You might find the system header file <*linux/if_arp.h*> useful for manipulating *ARP* request and reply messages, but remember that our version of these messages have an extra two-byte field as mentioned above.

6. **Your code should print out on *stdout,* in some appropriately clear format, the contents of the *Ethernet* frame header and *ARP* request message you send**. As described in Section 4.4 of the *ARP* & *RARP* handout, the node that responds to the request should, in its reply message, swap the two *sender* addresses with the two *target* addresses, as well as, of course, echo back the extra *identification* field sent with the request. **The protocol at this responding node should print out, in an appropriately clear format, both the request frame (header and *ARP* message) it receives and the reply frame it sends. Similarly, the node that sent the request should print out the**

reply frame it receives. Finally, recall that the node issuing the request sends out a broadcast *Ethernet* frame, but the responding node replies with a unicast frame.

V. API specifications

1. The API is for communication between the *Tour* process and the *ARP* process. It consists of a single function, *areq*, implemented in the *Tour* module. *areq* is called by *send_v4* function of the application every time the latter want to send out an *ICMP echo request* message:

   ```
   int areq (struct sockaddr *IPaddr, socklen_t sockaddrlen, struct hwaddr *HWaddr);
   ```

   IPaddr contains the primary or alias *IP*address of a 'target' node on the *LAN* for which the corresponding hardware address is being requested.

   hwaddr is a new structure (and not a pre-existing type) modeled on the *sockaddr_ll* of *PF_PACKET*; you will have to declare it in your code. It is used to return the requested hardware address to the caller of *areq* :

   ```
   structure hwaddr {
                        int             sll_ifindex;        /* Interface number */
                        unsigned short  sll_hatype;         /* Hardware type */
                        unsigned char   sll_halen;          /* Length of address */
                        unsigned char   sll_addr[8];        /* Physical layer address */
   };
   ```

2. *areq* creates a *Unix* domain socket of type *SOCK_STREAM* and connects to the 'well-known' *sun_path* file of the *ARP* listening socket. It sends the *IP* address from parameter *IPaddr* and the information in the three fields of parameter *HWaddr* to *ARP*. It then blocks on a read awaiting a reply from *ARP*. This read should be backed up by a timeout since it is possible that no reply is received for the request. If a timeout occurs, *areq* should close the socket and return to its caller indicating failure (through its *int* return value).

   Your application code should print out on *stdout*, in some appropriately clear format, a notification every time *areq* is called, giving the *IP* address for which a *HW* address is being sought. It should similarly print out the result when the call to *areq* returns (*HW address* returned, or failure).

3. When the *ARP* module receives a request for a *HW* address from *areq* through its *Unix* domain listening socket, it first checks if the required *HW* address is already in the cache. If so, it can respond immediately to the *areq* and close the *Unix* domain connection socket. Else : it makes an 'incomplete' entry in the cache, consisting of parts (i), (iii), (iv) and (v) ; puts out an *ARP* request message on the network on its *PF_PACKET* socket; and starts monitoring the *areq* connection socket for readability – if the *areq* client closes the connection socket (this would occur in response to a timeout in *areq*), *ARP* deletes the corresponding incomplete entry from the cache (and ignores any subsequent *ARP* reply from the network if such is received). On the other hand, if *ARP* receives a reply from the network, it updates the incomplete cache entry, responds to *areq*, and closes the connection socket.

Hand-in

Important: - The criterion for a successful assignment is that it compiles successfully on minix, and executes correctly on the minix VM environment; with clear, well-structured output that convinces us that the mechanisms you implemented are working correctly, and good documentations.

You can only submit the source code and a ReadMe file! No executable or object file is accepted; including them will result in deducted marks. Source code includes only .c files, .h files, and a Makefile. with those names. They must be archived and compressed into a .tar.gz file instead of multiple files. In your ReadMe file, include what command to use to obtain the source files from the .tar.gz file. Note that this command must run properly on minix.

You should submit your code using Blackboard. Your *Makefile*

- compiles your code using, where necessary, the Stevens' environment in the course account on the *minix* node, */home/users/cse533/Stevens/unpv13e* ; and

- gives standard names *tour_<login>*, arp_*<login>* for the executables produced (note the underscore in the executable names), where *<login>* is the login name your group uses to hand in its copy of the assignment.

In addition to command to decompress your .tar.gz file, the *ReadMe* file should contain is an identification of the members in the group. It also should include all major designs and decisions that you would need to remind yourself later on when trying to re-use your program again.

Aside from the ReadMe file, your should also include *program documentation*. It refers to the comments in your source code. Your submitted code should have a good amount of program documentation, to explain what each chunk of code does, the meaning and purpose of each field in a struct, and the meaning of each value an important variable can take.

Each group hands in just one copy of the Assignment, under one and only one member's login name. You must co-ordinate among your group so that only one copy is submitted. A penalty will be exercised if more than one copy is submitted per group.

Blackboard submission only - The handing-in will be through Blackboard Assignments. Note that there are more than one types of assignments supported by Blackboard. We do not use "SafeAssign" or "Digital Dropbox". Rather we use "Assignments" only. The submission instructions are at: http://it.stonybrook.edu/help/kb/creating-and-managing-assignments-in-blackboard. You must read the submission instructions very carefully, and check to make sure your assignment has been submitted correctly before the deadline. You can only submit once! However you can save your work by clicking "Save" as many times as you like. Only click "Submit" after you have checked and are certain that all requirements are followed.

Grouping

The assignment is to be completed by the same groups as in Assignments 2 and 3. The information on "Working in groups" is the same as in Assignment 2. Please see A2 specification for details.

Completing the assignment

Warning: You have less than two weeks to finish this assignment. Given the tight schedule for the final exam, You must start working on it right away.

Incremental development - You are adviced to develop your programs **incrementally** - starting from a simple version, and then add the various functionalities and capabilities one at a time.

Piazza discussion board - Read the man pages for various system calls and library functions. Reading the related textbook pages is must. You should make use of the Piazza discussion board to ask and answer questions. A forum called **Assignment 4** has beeen created for this purpose. It will be read by the teaching staff and all students. Note that no exact answer to any question should be provided at any time by any student or teaching staff, online or offline. Hints can be provided but not the exact answers.

Moss and academic integrity Your submitted code will be checked using the Stanford Moss package - *A System for Detecting Software Plagiarism*, to detect cheating. Source code from previous years of this course, which has all been archived, and from the current offering, i.e., your classmates, will be checked against by this package. Also keep in mind that important assignment materials are within the scope of the exam. So you must understand the major steps and knowledge elements of the assignment.

The due date is 11:55pm on Sunday, December 6. No late submissions will be accepted.

Last update: November 24, 2015