CSE 533 (Fall 2015) - Assignment 3

Due date: Monday, Nov. 23, at 11:55pm

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

1. Overview

In this assignment you will develop and implement :

- An **O**n-**D**emand shortest-hop **R**outing (*ODR*) protocol for routing messages in networks of fixed but arbitrary and unknown connectivity, using *PF_PACKET* sockets. The implementation is based on (a simplified version of) the *AODV* algorithm.

- A client/server time application in which clients and servers communicate with each other across a network; the messages exchanged are transmitted using *ODR*.

- An *API* that enables applications to communicate with the *ODR* mechanism running locally at their nodes, using *Unix* domain datagram sockets.

The following should prove useful reference material for the assignment :

- Sections 15.1, 15.2, 15.4 & 15.6, Chapter 15, on *Unix* domain datagram sockets.

- The PF_PACKET(7) slides (see `PF_PACKET(7).pdf` in folder) from the *Linux* manual pages.

- A note (see `PF_Packet_Useful_Link.pdf` in folder) from a past CSSE533 student, which points to a reference page on raw Ethernet programming, please skip the section titled "UDP".
  Also, the following link http://www.pdbuchan.com/rawsock/rawsock.html contains useful code samples that use PF_PACKET sockets (as well as other code samples that use raw IP sockets which you do not need for this assignment, though you will be using these types of sockets for Assignment 4).

- Charles E. Perkins & Elizabeth M. Royer. "Ad-hoc On-Demand Distance Vector Routing", Proceedings of the 2nd *IEEE* Workshop on Mobile Computing Systems and Applications, New Orleans, Louisiana, February 1999, *pp*. 90 - 100. (see `aodv.pdf` in folder)

2. The *VMware* environment

Your implementation will be run and tested in a VMware computing environment. *minix.cs.sunysb.edu* is a *Linux* machine running *VMware*. A cluster of ten *Linux* virtual machines, called *vm1* through *vm10*, on which you can gain access as *root* and run your code have been created on *minix*. *VMware* instructions (see `vmware.pdf` in folder) explain how to use the system. The ten virtual machines have been configured into a small virtual *intranet* of *Ethernet LAN*s whose topology is (in principle) unknown to you.

There is a course account *cse533* on *minix*, with home directory */users/cse533*. There you will find a subdirectory *Stevens/unpv13e* , exactly as you are used to having on the *cs* system. You should develop your source code and *makefile* on this machine. Note that you are not allowed to have any source code on the 10 virtual machines.

On minix, you do not need to link against the socket library (*-lsocket*) in *Linux*. The same is true for *-lnsl* and *-lresolv*. For example, take a look at how the *LIBS* variable is defined for Solaris, in */home/courses/cse533*

/Stevens/unpv13e_solaris2.10/Make.defines (on *compserv1*, say) :

LIBS = ../libunp.a -lresolv -lsocket -lnsl -lpthread

But if you take a look at *Make.defines* on *minix* (*/users/cse533/Stevens/unpv13e/Make.defines*) you will find only:

LIBS = ../libunp.a -lpthread

The nodes *vm1 , ... , vm10* are all multihomed : each has two (or more) interfaces. **The interface *'eth0'* should be completely ignored and is not to be used for this assignment** (because it shows all ten nodes as if belonging to the same single *Ethernet* 192.168.1.0/24, rather than to an *intranet* composed of several *Ethernet*s).

Note that *vm1 , ... , vm10* are virtual machines, not real ones. One implication of this is that you will not be able to find out what their (virtual) *IP* addresses are by using *nslookup* and such. To find out these *IP* addresses, you need to look at the file */etc/hosts* on *minix*. Moreover, invoking *gethostbyname* for a given *vm* will return to you only the (primary) *IP* address associated with the interface *eth0* of that *vm* (which is the interface you will not be using). It will not return to you any other *IP* address for the node. Similarly, *gethostbyaddr* will return the *vm* node name only if you give it the (primary) *IP* address associated with the interface *eth0* for the node. It will return nothing if you give it any other *IP* address for the node, even though the address is perfectly valid. Because of this, and because it will ease your task to be able to use *gethostbyname* and *gethostbyaddr* in a straightforward way, we shall adopt the (primary) *IP* addresses associated with interfaces *eth0* as the 'canonical' *IP* addresses for the nodes and use these IP addresses (not the *eth0* interfaces) for some purposes.

For help and documentation on *Linux*, make use of the man pages on *minix* as much as possible. E.g., you can get the man page for PF_PACKET using the command: *man 7 packet*.

3. The Time Application and the API

The time application implements the simple daytime service that we have seen in Chapter 1 of the textbook. A time server runs on each of the ten *vm* machines. The client code should also be available on each *vm* so that it can be evoked at any of them.

Normally, time clients/servers exchange request/reply messages using the *TCP*/*UDP* socket *API* that, effectively, enables them to receive service (indirectly, via the transport layer) from the local *IP* mechanism running at their nodes. You are to implement an *API* using *Unix* domain sockets to access the local *ODR* service directly, without any transport layer functions in the middle (somewhat similar, in effect, to the way that raw sockets permit an application to access *IP* directly). Use *Unix* domain ***SOCK_DGRAM***, rather than *SOCK_STREAM*, sockets (see Figures 15.5 & 15.6, *pp*. 418 - 419).

3.1 The API

You need to implement a *msg_send* function that will be called by clients/servers to send requests/replies. The parameters of the function consist of :

- *int* giving the socket descriptor for write
- *char** giving the 'canonical' *IP* address for the destination node, in presentation format
- *int* giving the destination 'port' number
- *char** giving the message to be sent
- *int flag* if set, force a route rediscovery to the destination node even if a non-'stale' route already exists (see below)

msg_send will format these parameters into a single *char* sequence which is written to the *Unix* domain socket that a client/server process creates. The sequence will be read by the local *ODR* from a *Unix* domain socket that the *ODR* process creates for itself.

Similarly, we need a *msg_recv* function which will do a (blocking) read on the application's *Unix* domain socket and return with :

- *int* giving socket descriptor for read
- *char** giving message received
- *char** giving 'canonical' *IP* address for the source node of message, in presentation format
- *int** giving source 'port' number

This information is written as a single *char* sequence by the *ODR* process to the domain socket that it creates for itself. It is read by *msg_recv* from the domain socket the peer client/server process creates, decomposed into the three components above, and returned to the caller of *msg_recv*.

Also see Section 5 below entitled: *ODR and the API*.

3.2 Client

When a client is evoked at a node, it creates a *Unix* domain datagram socket.

> The client should bind its socket to a 'temporary' (*i.e.*, not 'well-known') *sun_path* name obtained from a call to *tmpnam*() (*cf*. line 10, Figure 15.6, *p*. 419) so that multiple clients may run at the same node.
>
> Note that *tmpnam*() is actually highly deprecated. You should use the *mkstemp*() function instead - look up the online man pages on *minix* ('man mkstemp') for details.
>
> As you run client code again and again during the development stage, the temporary files created by the calls to *tmpnam* / *mkstemp* start to proliferate since these files are not automatically removed when the client code terminates. You need to explicitly remove the file created by the client evocation by issuing a call to *unlink*() or to *remove*() in your client code just before the client code exits. See the online man pages on *minix* ('man unlink', 'man remove') for details.

The client then enters an infinite loop repeating the steps below.

1. The client prompts the user to choose one of *vm1* , ... , *vm10* as a server node.

2. Client *msg_send*s a 1 or 2 byte message to server and prints out on *stdout* the message
   client at node vm i_1 sending request to server at vm i_2
   (In general, throughout this assignment, "trace" messages such as the one above should give the *vm* names and not *IP* addresses of the nodes.)

3. Client then blocks in *msg_recv* awaiting response. This attempt to read from the domain socket should be backed up by a timeout in case no response ever comes. I leave it up to you whether you 'wrap' the call to *msg_recv* in a timeout, or you implement the timeout inside *msg_recv* itself. When the client receives a response it prints out on *stdout* the message
   client at node vm i_1 : received from vm i_2 <timestamp>
   If, on the other hand, the client times out, it should print out the message
   client at node vm i_1 : timeout on response from vm i_2
   The client then retransmits the message out, setting the *flag* parameter in *msg_send* to force a route rediscovery, and prints out an appropriate message on *stdout*. For simplicity, this is done only once, when a timeout for a given message to the server occurs for the first time.

4. Client repeats steps 1. - 3.

3.3 Server

The server creates a *Unix* domain datagram socket. The server socket is assumed to have a (node-local) 'well-known' *sun_path* name which it *bind*s to. This *sun_path* should be unique to your group. This 'well-known' *sun_path* name is also designated by a (network-wide) 'well-known' 'port' value. The time client uses this 'port' value to communicate with the server.

The server enters an infinite sequence of calls to *msg_recv* followed by *msg_send*, awaiting client requests and responding to them. When it responds to a client request, it prints out on *stdout* the message

server at node vm i_1 responding to request from vm i_2

4. ODR

The *ODR* process runs on each of the ten *vm* machines. It is evoked with a single command line argument which gives a "*staleness*" time parameter, in seconds.

- It uses *get_hw_addrs* (available to you on *minix* in ~cse533/Asgn3_code) to obtain the index, and associated (unicast) *IP* and *Ethernet* addresses for each of the node's interfaces, except for the *eth0* and *lo* (loopback) interfaces, which should be ignored.

  In the subdirectory *~cse533/Asgn3_code* (*/users/cse533/Asgn3_code*) on *minix* I am providing you with two functions, *get_hw_addrs* and *prhwaddrs*. These are analogous to the *get_ifi_info_plus* and *prifinfo_plus* of Assignment 2. Like *get_ifi_info_plus*, *get_hw_addrs* uses *ioctl*. *get_hw_addrs* gets the (primary) *IP* address, alias *IP* addresses (if any), *HW* address, and interface name and index value for each of the node's interfaces (including the loopback interface *lo*). *prhwaddrs* prints that information out. You should modify and use these functions as needed.

  Note that if an interface has no *HW* address associated with it (this is, typically, the case for the loopback interface *lo* for example), then *ioctl* returns *get_hw_addrs* a *HW* address which is the equivalent of 00:00:00:00:00:00. *get_hw_addrs* stores this in the appropriate field of its data structures as it would with any *HW* address returned by *ioctl*, but when *prhwaddrs* comes across such an address, it prints a blank line instead of its usual '*HWaddr = xx*:*xx*:*xx*:*xx*:*xx*:*xx*'.

- The *ODR* process creates one or more *PF_PACKET* sockets.

  You will need to try out *PF_PACKET* sockets for yourselves and familiarize yourselves with how they behave. If, when you read from the socket and provide a sockaddr_ll structure, the kernel returns to you the index of the interface on which the incoming frame was received, then one socket will be enough. Otherwise, somewhat in the manner of Assignment 2, you shall have to create a *PF_PACKET* socket for every interface of interest (which are all the interfaces of the node, excluding interfaces *lo* and *eth0*), and bind a socket to each interface. Furthermore, if the kernel also returns to you the source Ethernet address of the frame in the sockaddr_ll structure, then you can make do with *SOCK_DGRAM* type *PF_PACKET* sockets; otherwise you shall have to use *SOCK_RAW* type sockets (although I would prefer you to use *SOCK_RAW* type sockets anyway, even if it turns out you can make do with *SOCK_DGRAM* type).

  The socket(s) should have a *protocol* value no larger than 0xffff (so that it fits in two bytes, given as a network-byte-order parameter in the call(s) to function *socket*) that identifies your *ODR* protocol. The <*linux/if_ether.h*> include file (*i.e.*, the file */usr/include/linux/if_ether.h*) contains *protocol* values defined for the standard protocols typically found on an *Ethernet LAN*, as well as other values such as *ETH_P_ALL*. You should set *protocol* to a value of your

choice which is **not** a <*linux/if_ether.h*> value, but which is, hopefully, unique to your group.

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 frames. For that reason, try to choose a *protocol* value for the socket(s) that is likely to be unique to your group (something based on your Stony Brook student ID number, for example). This value effectively becomes the *protocol* value for your implementation of *ODR*, as opposed to some other *cse533* student's implementation.

Because your value of *protocol* is to be carried in the *frame type* field of the *Ethernet* frame header, **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.

Note from the man pages for *packet*(7) that frames are passed to and from the socket without any processing in the frame content by the device driver on the other side of the socket, except for calculating and tagging on the 4-byte *CRC* trailer for outgoing frames, and stripping that trailer before delivering incoming frames to the socket. Nevertheless, if you write a frame that is less than 64 bytes, the necessary padding is automatically added by the device driver so that the frame that is actually transmitted out is the minimum *Ethernet* size of 64 bytes.

When reading from the socket, however, any such padding that was introduced into a short frame at the sending node to bring it up to the minimum frame size is **not** stripped off - it is included in what you receive from the socket (thus, the minimum number of bytes you receive should never be less than 64). Also, you will have to build the frame header for outgoing frames yourselves. Bear in mind that the field values in that header have to be in network order.

- The *ODR* process also creates a *Unix* domain datagram socket for communication with application processes at the node, and binds the socket to a 'well known' *sun_path* name for the *ODR* service. This *sun_path* should be unique to your group.

Because it is dealing with fixed topologies, *ODR* is, by and large, considerably simpler than *AODV*. In particular, discovered routes are relatively stable and there is no need for all the paraphernalia that goes with the possibility of routes changing (such as maintenance of active nodes in the routing tables and timeout mechanisms; timeouts on reverse links; *lifetime* field in the *RREP* messages; etc.)

Nor will we be implementing *source_sequence_#* field in the *RREQ* messages; and *dest_sequence_#* fields in *RREQ* and *RREP* messages. In reality, we should (though we will not, for the sake of simplicity, be doing so) implement some sort of sequence number mechanism, or some alternative mechanism such as split-horizon for example, if we are to avoid possible scenarios of routing loops in a "count to infinity" context (I shall explain this point in class).

However, we want *ODR* to discover shortest-hop paths, and we want it to do so in a reasonably efficient manner. This necessitates having one or two aspects of its operations work in a different, possibly slightly more complicated, way than *AODV* does. *ODR* has several basic responsibilities :

- Build and maintain a routing table. For each destination in the table, the routing table structure should include, at a minimum, the next-hop node (in the form of the *Ethernet* address for that node) and outgoing interface index, the number of hops to the destination, and a timestamp of when the the routing table entry was made or last "reconfirmed" / updated. Note that a destination node in the table is to be identified **only** by its 'canonical' *IP* address, and not by any other *IP* addresses the node has.

- Generate a *RREQ* in response to a time client calling *msg_send* for a destination for which *ODR* has no

route (or for which a route exists, but *msg_send* has the *flag* parameter set or the route has gone 'stale' – see below), and 'flood' the RREQ out on all the node's interfaces (except, of course, for interfaces *eth0* and *lo*).

Flooding is done using an *Ethernet* broadcast destination address (0xff:ff:ff:ff:ff:ff) in the outgoing frame header.

Note that a copy of the broadcast packet is supposed to / might be looped back to the node that sends it (see *p*. 535 in Stevens' textbook). *ODR* will have to take care not to treat these copies as new incoming *RREQ*s.

Also note that *ODR* at the client node increments the *broadcast_id* every time it issues a new *RREQ* for any destination node.

- When a *RREQ* is received, *ODR* has to generate a *RREP* if it is at the destination node, or if it is at an intermediate node that happens to have a route (which is not 'stale' – see below) to the destination. Otherwise, it must propagate the *RREQ* by flooding it out on all the node's interfaces (except the interface the *RREQ* arrived on).

  Note that as it processes received *RREQ*s, *ODR* should enter the 'reverse' route back to the source node into its routing table, or update an existing entry back to the source node if the *RREQ* received shows a shorter-hop route, or a route with the same number of hops but going through a different neighbour. The timestamp associated with the table entry should be updated whenever an existing route is either "reconfirmed" or updated. Obviously, if the node is going to generate a *RREP*, updating an existing entry back to the source node with a more efficient route, or a same-hops route using a different neighbour, should be done before the *RREP* is generated.

  Unlike *AODV*, when an intermediate node receives a *RREQ* for which it generates a *RREP*, it should nevertheless continue to flood the *RREQ* it received if the *RREQ* pertains to a source node whose existence it has heretofore been unaware of, or the *RREQ* gives it a more efficient route than it knew of back to the source node (the reason for continuing to flood the *RREQ* is so that other nodes in the *intranet* also become aware of the existence of the source node or of the potentially more optimal reverse route to it, and update their tables accordingly). However, since an *RREP* for this *RREQ* is being sent by our node, we do not want other nodes who receive the *RREQ* propagated by our node, and who might be in a position to do so, to also send *RREP*s. So we need to introduce a field in the *RREQ* message, not present in the *AODV* specifications, which acts like a "*RREP already sent*" field. Our node sets this field before further propagating the *RREQ* and nodes receiving an *RREQ* with this field set do not send *RREP*s in response, even if they are in a position to do so.

  ODR may, of course, receive multiple, distinct instances of the **same** *RREQ* (the combination of *source_addr* and *broadcast_id* uniquely identifies the *RREQ*). Such *RREQ*s should not be flooded out unless they have a lower hop count than instances of that *RREQ* that had previously been received.

  By the same token, if *ODR* is in a position to send out a *RREP*, and has already done so for this, now repeating, *RREQ* , it should not send out another *RREP* unless the *RREQ* shows a more efficient, previously unknown, reverse route back to the source node. In other words, *ODR* should not generate essentially duplicative *RREP*s, nor generate *RREP*s to instances of *RREQ*s that reflect reverse routes to the source that are not more efficient than what we already have.

- Relay *RREP*s received back to the source node (this is done using the 'reverse' route entered into the

routing table when the corresponding *RREQ* was processed). At the same time, a 'forward' path to the destination is entered into the routing table. *ODR* could receive multiple, distinct *RREP*s for the same *RREQ*. The 'forward' route entered in the routing table should be updated to reflect the shortest-hop route to the destination, and *RREP*s reflecting suboptimal routes should not be relayed back to the source.

In general, maintaining a route and its associated timestamp in the table in response to *RREP*s received is done in the same manner described above for *RREQ*s.

- Forward time client/server *application payload* messages along the next hop.

  (The following is important – you will lose points if you do not implement it.)
  Note that such *application payload messages* (especially if they are the initial time request from the client to the server, rather than the server response back to the client) can be like "free" *RREQ*s, enabling nodes along the path from source (client) to destination (server) node to build a reverse path back to the client node whose existence they were previously unaware of (or, possibly, to update an existing route with a more optimal one).

  Before it forwards an *application payload message* along the next hop, *ODR* at an intermediate node (and also at the final destination node) should use the message to update its routing table in this way. Thus, calls to *msg_send* by time servers should never cause *ODR* at the server node to initiate *RREQ*s, since the receipt of a time client request implies that a route back to the client node should now exist in the routing table. The only exception to this is if the server node has a *staleness* parameter of zero (see below).

- A routing table entry has associated with it a timestamp that gives the time the entry was made into the table. When a client at a node calls *msg_send*, and if an entry for the destination node already exists in the routing table, *ODR* first checks that the routing information is not '*stale*'. **A stale routing table entry** is one that is older than the value defined by the *staleness* parameter given as a command line argument to the *ODR* process when it is executed. *ODR* deletes stale entries (as well as non-stale entries when the *flag* parameter in *msg_send* is set) and initiates a route rediscovery by issuing a *RREQ* for the destination node. This will force periodic updating of the routing tables to take care of failed nodes along the current path, *Ethernet* addresses that might have changed, and so on.

  Similarly, as *RREQ*s propagate through the *intranet*, existing stale table entries at intermediate nodes are deleted and new route discoveries propagated. As noted above when processing *RREQ*s and *RREP*s, the associated timestamp for an existing table entry is updated in response to having the route either "reconfirmed" or updated (this applies to both reverse routes, by virtue of *RREQ*s received, and to forward routes, by virtue of *RREP*s).

  Finally, note that a *staleness* parameter of 0 essentially indicates that the discovered route will be used only once, when first discovered, and then discarded. Effectively, an *ODR* with *staleness* parameter 0 maintains no real routing table at all ; instead, it forces route discoveries at every step of its operation.

  As a practical matter, *ODR* should be run with *staleness* parameter values that are considerably larger than the longest *RTT* on the *intranet*, otherwise performance will degrade considerably (and collapse entirely as the parameter values approach 0). Nevertheless, for robustness, we need to implement a mechanism by which an intermediate node that receives a *RREP* or *application payload* message for forwarding and finds that its relevant routing table entry has since gone stale, can intiate a *RREQ* to rediscover the route it needs.

RREQ, *RREP*, and time client/server request/response messages will all have to be carried as encapsulated *ODR* protocol messages that form the data payload of *Ethernet* frames. So we need to design the structure of *ODR*

protocol messages. The format should contain a *type* field (0 for *RREQ*, 1 for *RREP*, 2 for *application payload*). The remaining fields in an *ODR* message will depend on what type it is. The fields needed for (our simplified versions of *AODV*'s) *RREQ* and *RREP* should be fairly clear to you, but keep in mind that you need to introduce two extra fields:

- The "*RREP already sent*" bit or field in *RREQ* messages, as mentioned above.
- A "*forced discovery*" bit or field in both *RREQ* and *RREP* messages:
    - When a client application forces route rediscovery, this bit should be set in the *RREQ* issued by the client node *ODR*.
    - Intermediate nodes that are not the destination node but which do have a route to the destination node should **not** respond with *RREP*s to an *RREQ* which has the *forced discovery* field set. Instead, they should continue to flood the *RREQ* so that it eventually reaches the destination node which will then respond with an *RREP*.
    - The intermediate nodes relaying such an *RREQ* **must** update their 'reverse' route back to the source node accordingly, even if the new route is less efficient (*i.e.*, has more hops) than the one they currently have in their routing table.
    - The destination node responds to the *RREQ* with an *RREP* in which this field is also set.
    - Intermediate nodes that receive such a *forced discovery RREP* **must** update their 'forward' route to the destination node accordingly, even if the new route is less efficient (*i.e.*, has more hops) than the one they currently have in their routing table.
    - This behaviour will cause a *forced discovery RREQ* to be responded to only by the destination node itself and not any other node, and will cause intermediate nodes to update their routing tables to both source and destination nodes in accordance with the latest routing information received, to cover the possibility that older routes are no longer valid because nodes and/or links along their paths have gone down.

A type 2, *application payload*, message needs to first contain the following types of information :

- *type* = 2
- 'canonical' *IP* address of source node
- 'port' number of source application process (This, of course, is not a real port number in the *TCP*/*UDP* sense, but simply a value that *ODR* at the source node uses to designate the *sun_path* name for the source application's domain socket.)
- 'canonical' *IP* address of destination node
- 'port' number of destination application process (This is passed to *ODR* by the application process at the source node when it calls *msg_send*. It designates the *sun_path* name for an application's domain socket at the destination node.)
- hop count (This starts at 0 and is incremented by 1 at each hop so that *ODR* can make use of the message to update its routing table, as discussed above.)
- number of bytes in application message

The fields above essentially constitute a 'header' for the *ODR* message. Note that fields which you choose to carry numeric values (rather than ascii characters, for example) must be in network byte order. *ODR*-defined numeric-valued fields in *type* 0, *RREQ*, and *type* 1, *RREP*, messages must, of course, also be in network byte order.

Also note that only the 'canonical' *IP* addresses are used for the source and destination nodes in the *ODR* header.

The same has to be true in the headers for *type* 0, *RREQ*, and *type* 1, *RREP*, messages. The general rule is that *ODR* messages only carry ‘canonical’ *IP* node addresses.

The last field in the *type* 2 *ODR* message is essentially the data payload of the message.

- application message given in the call to *msg_send*

An *ODR* protocol message is encapsulated as the data payload of an *Ethernet* frame whose header it fills in as follows :

- source address = *Ethernet* address of outgoing interface of the current node where *ODR* is processing the message.
- destination address = *Ethernet* broadcast address for *type* 0 messages; *Ethernet* address of next hop node for *type* 1 & 2 messages.
- *protocol* field = *protocol* value for the *ODR PF_PACKET* socket(s).

Last but not least, whenever *ODR* writes an *Ethernet* frame out through its socket, it prints out on *stdout* the message

ODR at node vm i_1: sending - frame hdr - src: vm i_1 dest: *addr*
ODR msg payload - message: *msg* type: *n* src: vm i_2 dest: vm i_3

where *addr* is in presentation format (*i.e.*, hexadecimal *xx*:*xx*:*xx*:*xx*:*xx*:*xx*) and gives the destination *Ethernet* address in the outgoing frame header. Other nodes in the message should be identified by their *vm* names. A message should be printed out for **each** packet sent out on a distinct interface. Note this means there will be ten windows that are open to display these messages, with one window per VM.

5. *ODR* and the *API*

When the *ODR* process first starts, it must construct a table in which it enters all well-known ‘port’ numbers and their corresponding *sun_path* names. These will constitute permanent entries in the table.

Thereafter, whenever it reads a message off its *Unix* domain socket, it must obtain the *sun_path* name for the peer process socket and check whether that name is entered in the table. If not, it must select an ‘ephemeral’ ‘port’ value by which to designate the peer *sun_path* name and enter the pair < port value, *sun_path* name > into the table.

Such entries cannot be permanently otherwise the table will grow unboundedly in time, with entries surviving forever, beyond the peer processes’ demise. We must associate a *time_to_live* field with a non-permanent table entry, and purge the entry if nothing is heard from the peer for that amount of time.

Every time a peer process for which a non-permanent table entry exists communicates with *ODR*, its *time_to_live* value should be reinitialized.

> Note that when *ODR* writes to a peer, it is possible for the write to fail because the peer does not exist : it could be a ‘ well-known’ service that is not running, or we could be in the interval between a process with a non-permanent table entry terminating and the expiration of its *time_to_live* value.

6. Notes

A proper implementation of *ODR* would probably require that *RREQ* and *RREP* messages be backed up by some kind of timeout and retransmission mechanism since the network transmission environment is not reliable. This would considerably complicate the implementation (because at any given moment, a node could have multiple *RREQ*s that it has flooded out, but for which it has still not received *RREP*s; the situation is further complicated by the fact that not all intermediate nodes receiving and relaying *RREQ*s necessarily lie on a path to the

destination, and therefore should expect to receive *RREP*s), and, learning-wise, would not add much to the experience you should have gained from Assignment 2.

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 documentation.

You can only submit the source code and the documentation! No executable or object file is accepted; including them will result in deducted marks. This means before you submit, you must make a clean submit directory that has only the required files in it. Source code includes only `.c` and `.h` files. The documentation includes two files: `Makefile` and `ReadMe`, with those names.

You should submit your code using Blackboard. Your submission must absolutely include a *Makefile* which:

- compiles your code using, where necessary, the Stevens' environment in the course account on the *minix* node, */users/cse533/Stevens/unpv13e* ; and
- gives standard names *ODR_<login>*, *server_<login>*, *client_<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.

Do not forget to hand in the *ReadMe* file. The first thing the *ReadMe* file should contain is an identification of the members in the group. You should include all major designs and decisions that you made for the client, server, and odr implementations.

Aside from the ReadMe file, the documentation also includes *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.

You may submit a .tar file instead of multiple files. In that event, you must include in your documentation what command to use to obtain the source files from the .tar file. Note that this command must run properly on minix.

Blackboard submission only - The handing-in will be through Blackboard Assignments. 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 Assignment 2. The information on "Working in groups" is the same as in Assignment 2. Please see A2 specification for details.

Completing the assignment

Warning: This is a very heavy and time-consuming assignment. You simply have to start working on it right away. Given the tight schedule for the next assignment, there will mostly likely be no extension for this assignment, thus it is imperative that you start 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 3** 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 steps and knowledge elements of the assignment.

The due date is 11:55pm on Monday, November 23. No late submissions will be accepted.

Last update: November 4, 2015