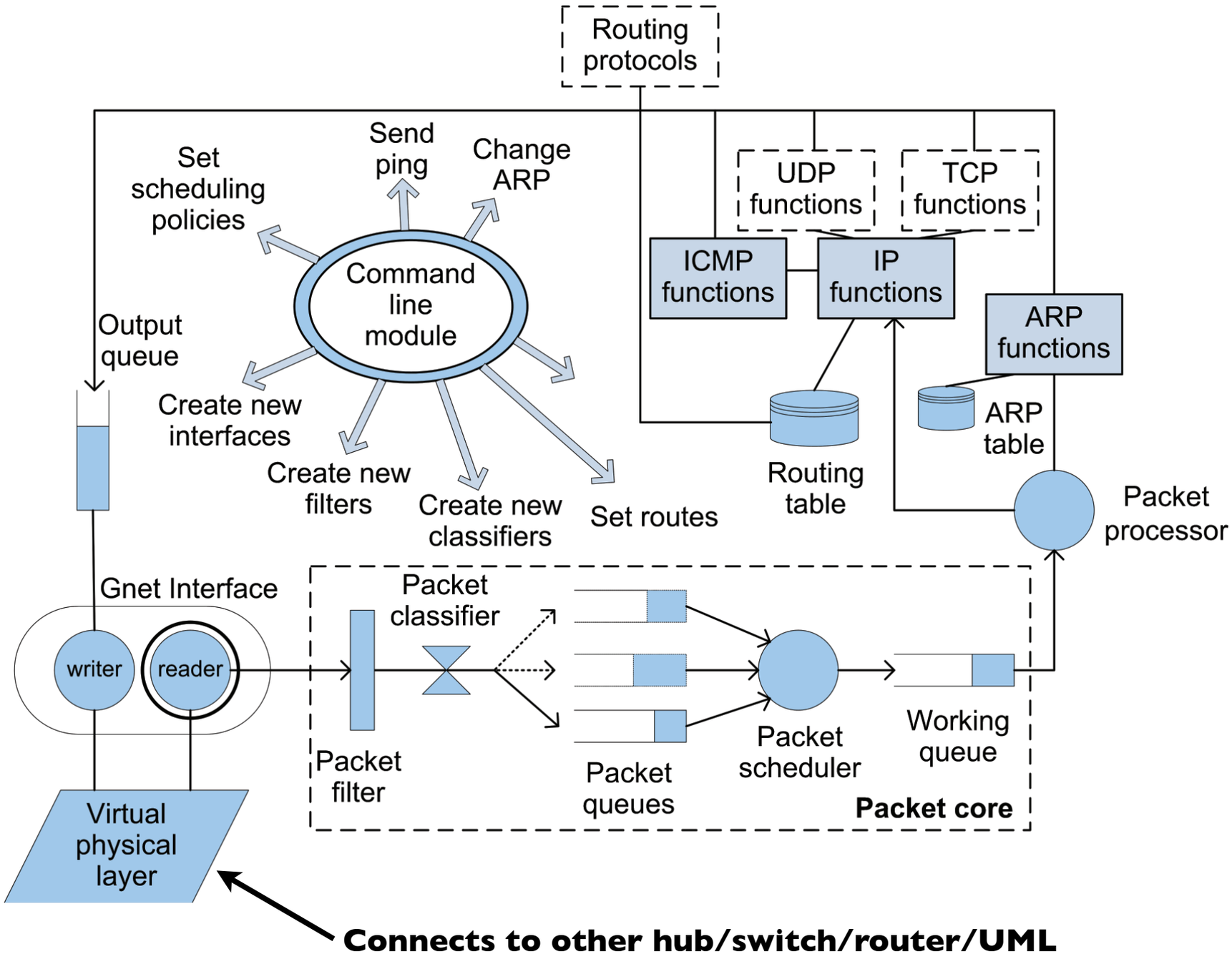
gRouter Programming

gRouter Architecture



Virtual Physical Layer

- Library of routines to connect with other network elements
  - UMLs
  - Routers
  - Switches
- Important functions provided:
  - vpl_connect(), vpl_recvfrom(), vpl_sendto(), vpl_connect()

Virtual Physical Layer...

vpl_sendto()

- Only function that injects packets into the "wire" (remember wire is actually a socket)
- Not directly used - unless there is a special need that cannot be accommodated by the existing packet flow
- gNet functions use this as the backend to write packets
- toEthernetDev() uses this function

Virtual Physical Layer...

vpl_recvfrom()

- This is the routine that is responsible for picking up the packet from the "wire"
- It is NOT directly called - it is used by fromEthernetDev() to get packets
- fromEthernetDev() runs on a thread allocated for each network interface

gPacket Structure

Each packet has its own gPacket structure when it lives within the gRouter - gPacket memory is de-alloced after the packet is written

```
typedef struct _gpacket_t
{
    pkt_frame_t frame;
    pkt_data_t data;
} gpacket_t;

// frame wrapping every packet... GINI specific (GINI me
typedef struct _pkt_frame_t
{
    int src_interface;                  // incoming interfa
    uchar src_ip_addr[4];               // source IP addres
    uchar src_hw_addr[6];               // source MAC addre
    int dst_interface;                  // outgoing interfa
    uchar nxth_ip_addr[4];              // destination inte
    int arp_valid;
    int arp_bcast;
} pkt_frame_t;

typedef struct _pkt_data_t
{
    struct
    {
        uchar dst[6];                   // destination host
        uchar src[6];                   // source host's MA
        ushort prot;                   // protocol field
    } header;
    uchar data[DEFAULT_MTU];                // payload (lim
} pkt_data_t;
```

gPacket Structure...

- **data:** holds the packet to be written or just read in
- **frame:** holds the meta data regarding the packet
  - Interface on which the packet arrived
  - ARP status - useful for the ARP resolver
- **frame:** mechanism to pass messages across gRouter modules (gRouter avoids or minimizes global variables)

gNet Device

- **gNet:** Network Interface in gRouter
  - Responsible for starting the network interfaces
  - **ethernet.c implements the interface drivers** toEthernetDev() **and** fromEthernetDev()
  - **creates, deletes, ups, and downs interfaces**
  - up interface: launches **thread to poll**

gNet Output Functions

- Reads output Queue
- Finds correct Interface
- Check ARP - if not delay
- If ARP, send packet

```
void *GNETHandler(void *outq)
{
     ...
     pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
     while (1)
     {
          if (readQueue(outputQ, (void **)&in_pkt, &inbytes) == EXIT_F
               return NULL;
          pthread_testcancel();
          if ((iface = findInterface(in_pkt->frame.dst_interface)) ==
          {
               error("[gnetHandler]:: Packet dropped, interface [%d] is
               continue;
          } else if (iface->state == INTERFACE_DOWN)
          {
               error("[gnetHandler]:: Packet dropped! Interface not up"
               continue;
          }
          // we have a valid interface handle -- iface.
          COPY_MAC(in_pkt->data.header.src, iface->mac_addr);
          if (in_pkt->frame.arp_valid == TRUE)
               putARPCache(in_pkt->frame.nxth_ip_addr, in_pkt->data.hea
          else if (in_pkt->frame.arp_bcast != TRUE)
          {
               if ((cached = lookupARPCache(in_pkt->frame.nxth_ip_addr,
                                 mac_addr)) == TRUE)
                    COPY_MAC(in_pkt->data.header.dst, mac_addr);
               else
               {
                    ARPResolve(in_pkt);
                    continue;
               }
          }
          iface->devdriver->todev((void *)in_pkt);

     }
}
```

gNet Input Functions

- upInterface runs the poll thread
- fromEthernetDev implements
  - packet reception
  - check if packet for router
  - create gPacket
  - apply filtering rules
  - enqueue packet in packet core

```
void* fromEthernetDev(void *arg)
{
        ..

    gpacket_t *in_pkt;

    pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);          //
    while (1)
    {
        vpl_recvfrom(iface->vpl_data, &(in_pkt->data), sizeof(pkt_data
        pthread_testcancel();
        // check whether the incoming packet is a layer 2 broadcast or

        if ((COMPARE_MAC(in_pkt->data.header.dst, iface->mac_addr) !=
            (COMPARE_MAC(in_pkt->data.header.dst, bcast_mac) != 0))
        {
            free(in_pkt);
            continue;
        }
        // copy fields into the message from the packet..
        in_pkt->frame.src_interface = iface->interface_id;
        COPY_MAC(in_pkt->frame.src_hw_addr, iface->mac_addr);
        COPY_IP(in_pkt->frame.src_ip_addr, iface->ip_addr);
        // check for filtering.. if the it should be filtered.. then d
        if (filteredPacket(filter, in_pkt))
        {
            free(in_pkt);
            continue;   // skip the rest of the loop
        }
        enqueuePacket(pcore, in_pkt, sizeof(gpacket_t));
    }
}
```

Packet Core

- Packet Core is responsible for supporting a bunch of packet queues
- Incoming packets are queued according to the classifiers they match
- Packets are picked from the queues and sent to the workQ
- Runs the packet scheduler

Inside ARP: Sending

- Sending side does ARP resolution
- ARP resolution can buffer the packet

```
/*
 * ARPResolve: this routine is responsible for local ARP resol
 * It consults the local ARP cache to determine whether a vali
 * is present. If a valid entry is not present, a remote reque
 * and the packet that caused the request is buffered. The buf
 * when the reply comes in.
 */
int ARPResolve(gpacket_t *in_pkt)
{
    uchar mac_addr[6];
    char tmpbuf[MAX_TMPBUF_LEN];

    in_pkt->data.header.prot = htons(IP_PROTOCOL);
    // lookup the ARP table for the MAC for next hop
    if (ARPFindEntry(in_pkt->frame.nxth_ip_addr, mac_addr) == E
    {
        // no ARP match, buffer and send ARP request for next
        verbose(2, "[ARPResolve]:: buffering packet, sending AR
        ARPAddBuffer(in_pkt);
        in_pkt->frame.arp_bcast = TRUE;
        // create a new message for ARP request
        ARPSendRequest(in_pkt);
        return EXIT_SUCCESS;;
    }

    verbose(2, "[ARPResolve]:: sent packet to MAC %s", MAC2Colo
    COPY_MAC(in_pkt->data.header.dst, mac_addr);
    in_pkt->frame.arp_valid = TRUE;
    ARPSend2Output(in_pkt);

    return EXIT_SUCCESS;
```

Inside ARP: Receiving

- At ARP arrival, *note ARP-IP binding*
- If ARP is meant for router, process it further
- Request or reply processing

```
void ARPProcess(gpacket_t *pkt)
{
    arp_packet_t *apkt = (arp_packet_t *) pkt->data.data;

    ARPAddEntry(gNtohl((uchar *)tmpbuf, apkt->src_ip_addr), a

    // Check it's actually destined to us,if not throw packet
    if (COMPARE_IP(apkt->dst_ip_addr, gHtonl((uchar *)tmpbuf,
    .....
    // We have a valid ARP packet, lets process it now.
    // If it's a REQUEST, send a reply back
    if (ntohs(apkt->arp_opcode) == ARP_REQUEST)
    {
        apkt->arp_opcode = htons(ARP_REPLY);
        COPY_MAC(apkt->src_hw_addr, pkt->frame.src_hw_addr);
        COPY_MAC(apkt->dst_hw_addr, pkt->data.header.src);
        COPY_IP(apkt->dst_ip_addr, apkt->src_ip_addr);
        COPY_IP(apkt->src_ip_addr, gHtonl((uchar *)tmpbuf, pk

        pkt->frame.dst_interface = pkt->frame.src_interface;

        COPY_MAC(pkt->data.header.dst, pkt->data.header.src);
        COPY_MAC(pkt->data.header.src,  pkt->frame.src_hw_add
        COPY_IP(pkt->frame.nxth_ip_addr, gNtohl((uchar *)tmpb
        pkt->frame.arp_valid = TRUE;

        pkt->data.header.prot = htons(ARP_PROTOCOL);

        ARPSend2Output(pkt);
    }
    else if (ntohs(apkt->arp_opcode) == ARP_REPLY)
        ARPFlushBuffer(gNtohl((uchar *)tmpbuf, apkt->src_ip_a
    ...
}
```

Inside ICMP: Receiving

- Processes incoming ICMP packets
- Echo requests need replies sent out

```
/*
 * send a PING reply in response to the incoming REQUEST
 */
void ICMPProcessEchoRequest(gpacket_t *in_pkt)
{
    ip_packet_t *ipkt = (ip_packet_t *)in_pkt->data.data;
    int iphdrlen = ipkt->ip_hdr_len *4;
    icmphdr_t *icmphdr = (icmphdr_t *)((uchar *)ipkt + iphdrlen);
    uchar *icmppkt_b = (uchar *)icmphdr;

    ushort cksum;
    int ilen = ntohs(ipkt->ip_pkt_len) - iphdrlen;

    icmphdr->type = ICMP_ECHO_REPLY;
    icmphdr->checksum = 0;
    if (IS_ODD(ilen))
    {
        // pad with a zero byte.. IP packet length remains the same
        icmppkt_b[ilen] = 0x0;
        ilen++;
    }
    cksum = checksum(icmppkt_b, (ilen/2));
    icmphdr->checksum = htons(cksum);

    // send the message back to the IP routine for further processing .
    // set the messsage as REPLY_PACKET..
    // destination IP and size need not be set. they can be obtained fr

    IPOutgoingPacket(in_pkt, NULL, 0, 0, ICMP_PROTOCOL);
}
```

Inside ICMP: Sending

- Ping request is an outgoing ICMP message
- Transport layer implementation will entail more ICMP message generation such as port unreachable

```
void ICMPSendPingPacket(uchar *dst_ip, int size, int seq)
{
    gpacket_t *out_pkt = (gpacket_t *) malloc(sizeof(gpacket_t));
    ip_packet_t *ipkt = (ip_packet_t *)(out_pkt->data.data);
        ....

    pstat.ntransmitted++;

    icmphdr->type = ICMP_ECHO_REQUEST;
    icmphdr->code = 0;
    icmphdr->checksum = 0;
    icmphdr->un.echo.id = getpid() & 0xFFFF;
    icmphdr->un.echo.sequence = seq;
    gettimeofday(tp, &tz);

    dataptr = ((uchar *)icmphdr + 8 +  sizeof(struct timeval));
    // pad data...
    for (i = 8; i < size; i++)
        *dataptr++ = i;

    cksum = checksum((uchar *)icmphdr, size/2);  // size = payload
    icmphdr->checksum = htons(cksum);

    verbose(2, "[sendPingPacket]:: Sending... ICMP ping to  %s", IP

    // send the message to the IP routine for further processing
    // the IP should create new header .. provide needed informatio
    // tag the message as new packet
    // IPOutgoingPacket(/context, packet, IPaddr, size, newflag, so
    IPOutgoingPacket(out_pkt, dst_ip, size, 1, ICMP_PROTOCOL);
}
```

Inside IP: Receiving

- If packet for router, process it further
- If not, check broadcasts (not implemented)
- If not, forward the packet

```
void IPIncomingPacket(gpacket_t *in_pkt)
{
    char tmpbuf[MAX_TMPBUF_LEN];
    // get a pointer to the IP packet
        ip_packet_t *ip_pkt = (ip_packet_t *)&in_pkt->data
    uchar bcast_ip[] = IP_BCAST_ADDR;

    // Is this IP packet for me??
    if (IPCheckPacket4Me(in_pkt))
    {
        verbose(2, "[IPIncomingPacket]:: got IP packet dest
        IPProcessMyPacket(in_pkt);
    } else if (COMPARE_IP(gNtohl(tmpbuf, ip_pkt->ip_dst), b
    {
        // TODO: rudimentary 'broadcast IP address' check
        verbose(2, "[IPIncomingPacket]:: not repeat broadca
                IP2Dot(tmpbuf, gNtohl((tmpbuf+20), ip_pkt->i
        IPProcessBcastPacket(in_pkt);
    } else
    {
        // Destinated to someone else
        verbose(2, "[IPIncomingPacket]:: got IP packet dest
        IPProcessForwardingPacket(in_pkt);
    }
}
```

Inside IP: Receiving...

- Packet is validated before forwarding, if not valid (e.g., expired TTL), ICMP is generated

```
int IPProcessForwardingPacket(gpacket_t *in_pkt)
{
      ...
      if (IPCheck4Errors(in_pkt) == EXIT_FAILURE)
            return EXIT_FAILURE;

      // find the route... if it does not exist, should we send a
      // ICMP network/host unreachable message -- CHECK??
      if (findRouteEntry(route_tbl, gNtohl(tmpbuf, ip_pkt->ip_dst),
                  in_pkt->frame.nxth_ip_addr,
                  &(in_pkt->frame.dst_interface)) == EXIT_FAILURE)
            return EXIT_FAILURE;

      IPCheck4Redirection(in_pkt);

      // check for fragmentation -- this should return three conditions:
      // FRAGS_NONE, FRAGS_ERROR, MORE_FRAGS
      need_frag = IPCheck4Fragmentation(in_pkt);

      switch (need_frag)
      {
      case FRAGS_ERROR:
            verbose(2, "[IPProcessForwardingPacket]:: unreachable on packet from %s
                  IP2Dot(tmpbuf, gNtohl((tmpbuf+20), ip_pkt->ip_src)));
            ICMPProcessFragNeeded(in_pkt);
            break;

      case MORE_FRAGS:
            // fragment processing...
            num_frags = fragmentIPPacket(in_pkt, pkt_frags);

            ...
            break;
      default:
            return EXIT_FAILURE;
      }
      return EXIT_SUCCESS;
}
```

Inside IP: Sending

- IP packet could be "new" or reply to an incoming packet
- Need to find the "source" IP for new packets

```
int IPOutgoingPacket(gpacket_t *pkt, uchar *dst_ip, int size, int newflag, int sr
{
    ...
    if (newflag == 0)
    {
        COPY_IP(ip_pkt->ip_dst, ip_pkt->ip_src);          // set dst to origina
        COPY_IP(ip_pkt->ip_src, gHtonl(tmpbuf, pkt->frame.src_ip_addr));    // s

        if (findRouteEntry(route_tbl, gNtohl(tmpbuf, ip_pkt->ip_dst),
                    pkt->frame.nxth_ip_addr, &(pkt->frame.dst_interface)) == EXI
                    return EXIT_FAILURE;
    } else if (newflag == 1)
    {
        ...
        COPY_IP(ip_pkt->ip_dst, gHtonl(tmpbuf, dst_ip));
        ip_pkt->ip_pkt_len = htons(size + ip_pkt->ip_hdr_len * 4);

        if (findRouteEntry(route_tbl, gNtohl(tmpbuf, ip_pkt->ip_dst),
                    pkt->frame.nxth_ip_addr, &(pkt->frame.dst_interface)) == EXI
                    return EXIT_FAILURE;

        verbose(2, "[IPOutgoingPacket]:: lookup MTU of nexthop");
        // lookup the IP address of the destination interface..
        if ((status = findInterfaceIP(MTU_tbl, pkt->frame.dst_interface,
                            iface_ip_addr)) == EXIT_FAILURE)
                            return EXIT_FAILURE;
    } else
    {
        error("[IPOutgoingPacket]:: unknown outgoing packet action.. packet disc
        return EXIT_FAILURE;
    }

    //   compute the new checksum
    cksum = checksum((uchar *)ip_pkt, ip_pkt->ip_hdr_len*2);
    ip_pkt->ip_cksum = htons(cksum);
    pkt->data.header.prot = htons(IP_PROTOCOL);

    IPSend2Output(pkt);
    verbose(2, "[IPOutgoingPacket]:: IP packet sent to output queue.. ");
    return EXIT_SUCCESS;
```

Extending gRouter

- As an example, lets extend the gRouter by adding UDP processing
- *.c files need to be added
- *.h files need to be added
- scons script needs patching
- build gRouter

Rebuilding gRouter

- Stop all gRouter instances
- Copy newly built gRouter
- Test gRouter by restarting the topologies