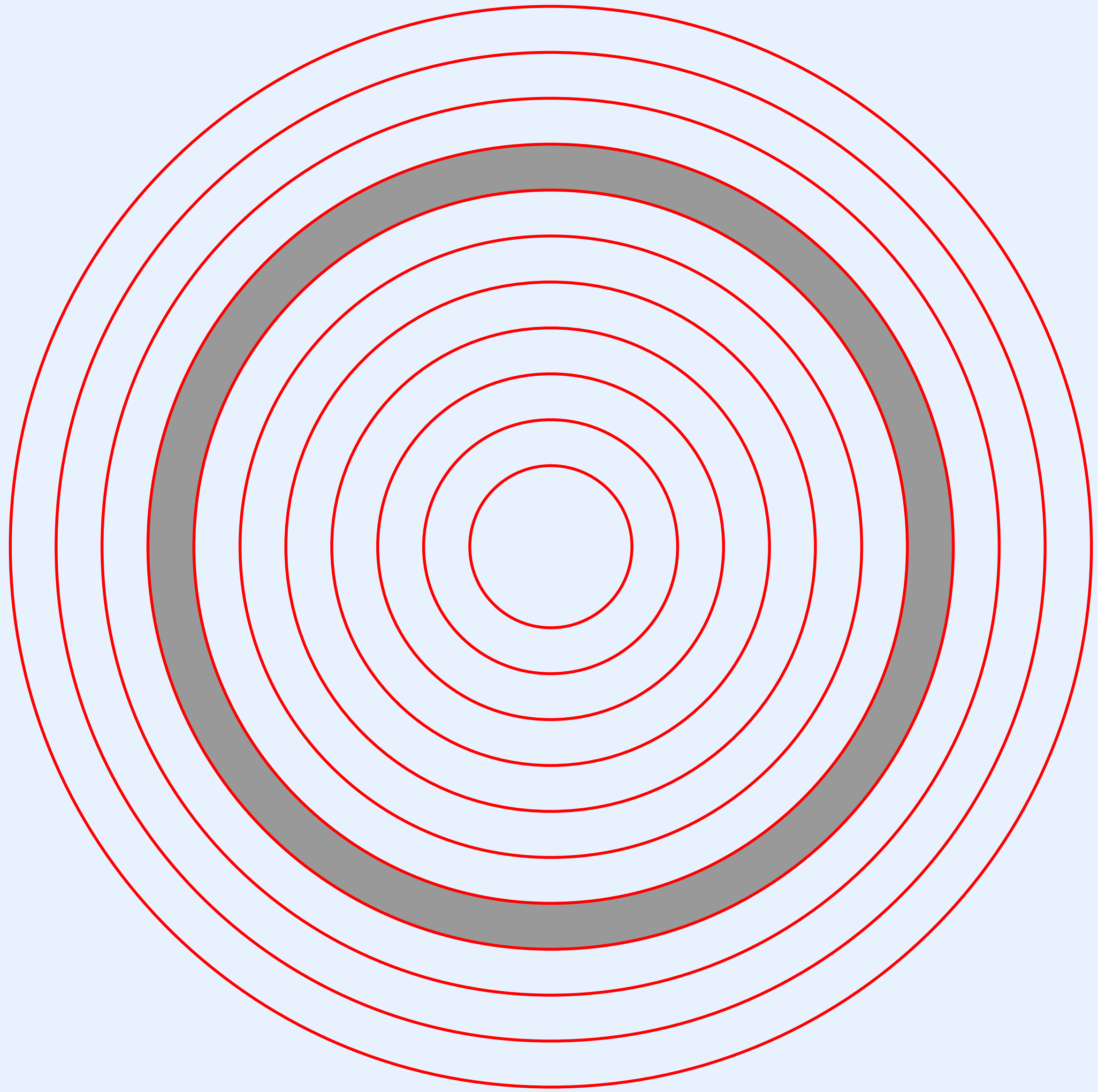
Module X

Networking And Protocol Implementation

Location Of Networking In The Hierarchy



Is The Hierarchical Level Correct?

- There are two possible approaches
  - Build a conventional operating system and add networking
  - Build networking code first and ensure all pieces of the operating system are distributed (e.g., a distributed process manager)
- Xinu places networking code at a high level of the hierarchy because most of the operating system is not distributed

A Fundamental Observation

One cannot undertake an operating system design without including network communication protocols, even in the embedded systems world.

Communication Systems

- A variety of network technologies have been devised
  - Wired (e.g., Ethernet)
  - Wireless (e.g., Wi-Fi and 5G)
- A computer can use
  - Local network communication: communicate directly over a network with other systems on the same network
  - Internet communication: communicate over a local network, but send packets through a *router* to an arbitrary computer on the Internet
- Internet communication has become the standard except for small, special-purpose embedded systems

Communication Protocols

- We use the term *communication protocols* to describe the standards that specify communication details such as
  – Message formats
  – Data representation (e.g., endinness)
  – Message exchange
  – How to handle errors
- Protocols used in the Internet are known as TCP/IP protocols

Communication Protocols And This Course

- We will
  - Not discuss the purpose of protocols
  - Consider only a few basic examples
  - Examine a minimalistic implementation
  - Omit many details
  - Focus on the API and the process model rather than the implementation
- To learn more
  - Read a leading text on TCP / IP
  - Take an internetworking course that uses an expert’s text

The Interface To Network Hardware

- As in most operating systems, Xinu provides an I/O device that can be used to transmit and receive packets
- For example, on a computer that includes Ethernet hardware, the Xinu device is usually named *ETHER*
- The device driver provides
  - Synchronous *read* that blocks until a packet arrives and then returns the packet
  - Synchronous *write* that blocks until a buffer is available and then accepts an outgoing packet
- We will assume all communication uses the Ethernet
- See Chapter 16 in text for explanation of how such a driver works

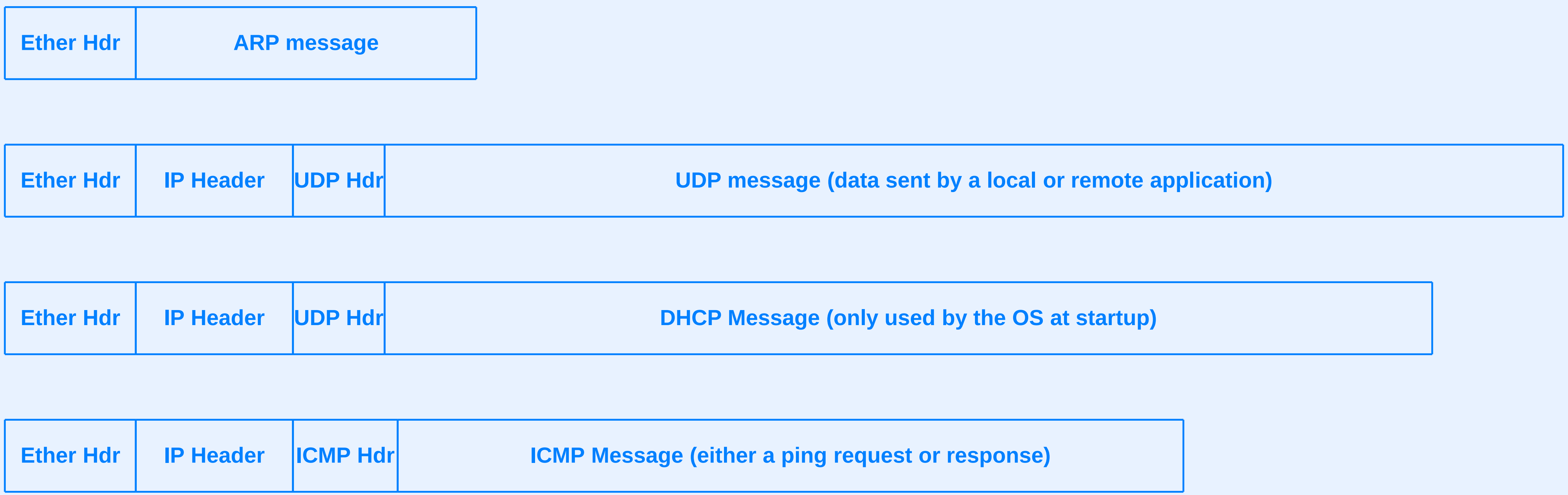
Protocols In Our Example

- You do not need to understand protocols, but you will see the following names

IP	Internet Protocol – defines an address for each computer on the global Internet and a header used to identify the sender and intended recipient for each packet
UDP	User Datagram Protocol – defines port numbers used to identify an application on a given computer and a header to specify them
ARP	Address Resolution Protocol – allows a computer to find the Ethernet address of a computer given its IP address
DHCP	Dynamic Host Configuration Protocol – used by a computer at startup to obtain an IP address and related information
ICMP	Internet Control Message Protocol – in our implementation, only used by the *ping* program to see if a computer is alive

Protocol Headers

- Internet protocols are *layered*, meaning headers are concatenated
- In our implementation a packet being sent or received will have one of following forms:



Implementing Concatenated Headers

- Most systems build a packet dynamically, adding headers one at a time as needed
- Xinu takes a shortcut: define two structures
  - One for an Ethernet header followed by an arp message
  - Another for the three cases of an Internet packet
    - Ethernet header, IP header, UDP header, UDP message
    - Ethernet header, IP header, UDP header, DHCP message
    - Ethernet header, IP header, ICMP header, ICMP message
- A further simplification: only *echo request* and *echo reply* types of ICMP messages are used (the *ping* program)
- The solution doesn't pretend to be general, but is *much* easier to understand than typical protocol software

Packet Format Declarations

- A single struct (*netpacket*) defines the three cases of an Internet packet
- The implementation assumes
  - Only an Ethernet network is used
  - There are no “options” present in the IP packet (a reasonable assumption)
- The netpacket struct defines an Ethernet packet header followed by an IP header, and then has a union to define
  - A UDP packet encapsulated in the Ethernet packet
  - An ICMP echo request or reply packet encapsulated in the Ethernet packet
- A separate structure is used to define ARP packets

Network Definitions In net.h (Part 1)

```
/* net.h */

#define NETSTK          8192            /* Stack size for network setup */
#define NETPRIO         500             /* Network startup priority     */
#define NETBOOTFILE     128             /* Size of the netboot filename */

/* Constants used in the networking code */

#define ETH_ARP     0x0806              /* Ethernet type for ARP        */
#define ETH_IP      0x0800              /* Ethernet type for IP         */
#define ETH_IPv6    0x86DD              /* Ethernet type for IPv6       */

/* Format of an Ethernet packet carrying IPv4 and UDP */

#pragma pack(2)
struct  netpacket       {
        byte    net_ethdst[ETH_ADDR_LEN];/* Ethernet dest. MAC address  */
        byte    net_ethsrc[ETH_ADDR_LEN];/* Ethernet source MAC address */
        uint16  net_ethtype;            /* Ethernet type field          */
        byte    net_ipvh;               /* IP version and hdr length    */
        byte    net_iptos;              /* IP type of service           */
        uint16  net_iplen;              /* IP total packet length       */
        uint16  net_ipid;               /* IP datagram ID               */
        uint16  net_ipfrag;             /* IP flags & fragment offset   */
        byte    net_ipttl;              /* IP time-to-live              */
        byte    net_ipproto;            /* IP protocol (actually type)  */
        uint16  net_ipcksum;            /* IP checksum                  */
        uint32  net_ipsrc;              /* IP source address            */
        uint32  net_ipdst;              /* IP destination address       */
```

Network Definitions In net.h (Part 2)

```
        union {
         struct {
          uint16        net_udpsport;   /* UDP source protocol port     */
          uint16        net_udpdport;   /* UDP destination protocol port*/
          uint16        net_udplen;     /* UDP total length             */
          uint16        net_udpcksum;   /* UDP checksum                 */
          byte          net_udpdata[1500-28];/* UDP payload (1500-above)*/
         };
         struct {
          byte          net_ictype;     /* ICMP message type            */
          byte          net_iccode;     /* ICMP code field (0 for ping) */
          uint16        net_iccksum;    /* ICMP message checksum        */
          uint16        net_icident;    /* ICMP identifier              */
          uint16        net_icseq;      /* ICMP sequence number         */
          byte          net_icdata[1500-28];/* ICMP payload (1500-above)*/
         };
        };
};

#pragma pack()

#define PACKLEN sizeof(struct netpacket)

extern  bpid32  netbufpool;             /* ID of net packet buffer pool */
```

Network Definitions In net.h (Part 3)

```
struct  network {
        uint32  ipucast;
        uint32  ipbcast;
        uint32  ipmask;
        uint32  ipprefix;
        uint32  iprouter;
        uint32  bootserver;
        bool8   ipvalid;
        byte    ethucast[ETH_ADDR_LEN];
        byte    ethbcast[ETH_ADDR_LEN];
        char    bootfile[NETBOOTFILE];
};

extern  struct  network NetData;        /* Local Network Interface      */
```

- Variable *NetData* holds network information obtained at startup, including
  - The computer’s IP address
  - The address mask for other local network
  - The address of an Internet router

Services An Application Can Use

- In this version of Xinu, an application can either
  - Use UDP to exchange messages with another application running on a computer on the Internet
  - Use ICMP to send a *ping* packet and receive a reply from an arbitrary computer on the Internet
- The other protocols (ARP and DHCP) merely provide support; they are invisible to an application

Identifying An Application

- UDP allows multiple applications on a given computer to communicate with other applications running on computers attached to the Internet
- To identify a destination application, a sending application must specify two items
  - The computer on which the destination runs
  - An ID that identifies a specific application
- For the two items, UDP uses
  - The 32-bit IP address of a computer
  - A 16-bit port integer called a *port number* that identifies an application
- For this course, you do not need to know how IP addresses and port numbers are obtained, just understand that two items are needed to identify each application

Features Of Networks Related To Operating Systems

- Two aspects of Internet software relate directly to the operating system
  - The interface that the operating system supplies to applications
  - The process structure used internally to implement protocols
- We will consider both aspects

Our Example API

- Is analogous to the socket API used by many operating systems (but all the details differ)
- Works as follows
  - Before sending data, an application must call a function to register endpoint information
  - The operating system responds by issuing a small integer descriptor to be used for communication (informally, we call the descriptor a *slot number*)
  - The application uses the descriptor to *send* and *receive* data
  - When finished, the application releases the descriptor

UDP Functions That Xinu Supplies To Applications

- *udp_register* – called by an application to register endpoint information, a remote (IP, port) tuple and a local UDP port
- *udp_send* - called by an application to send a UDP packet to a previously-registered endpoint
- *udp_recv* - called by an application to receive a UDP packet from a previously-registered remote endpoint
- *udp_recvaddr* - called by an application to receive a UDP packet and record the sender's address (allows an application to receive messages from an arbitrary application)
- *udp_release* - called by an application to release a previously-registered endpoint
- Notes: udp_register returns a descriptor (slot number) that must be passed to the other functions

When An Invalid Packet Arrives

- When UDP packet arrives, the network code calls internal function *udp_in*
- *Udp_in* searches table of registered endpoints
- If the incoming packet matches a registered endpoint, the packet is enqueued on the entry, and if a process is waiting, the process becomes ready and reads the message
- If no match is found, the incoming packet is ignored (silently dropped)

Timeout And Retransmission

- Retransmission of a packet is fundamental in networking
- Retransmission handles packet loss by sending a duplicate copy
- Typically, the side sending a request performs retransmission
- The steps taken
  - Send a request and wait *N* milliseconds for a reply
  - If no reply comes, send the request again
  - Repeat sending and waiting *K* times before declaring failure (*K* is usually a small number, such as 3)

Timeout In Xinu

- Xinu uses *recvtime* (receive-with-timeout) to wait for a specified time
- A call to *recvtim* is incorporated into *udp_recv*
- An application
  - Calls *udp_recv* to wait for an incoming packet
  - Specifies a timeout as an argument
- *Udp_recv*
  - Calls *recvtime* to wait for a packet
  - Returns a packet if one arrives before the specified timeout
  - Returns *TIMEOUT* otherwise
- The application can choose to resend the request and wait again

An Example Of Using UDP

- Send a message to port UDP 37 on computer 128.10.3.8 and receive a reply, retrying once
  - Choose a local UDP port (*TIMELPORT* in the code)
  - Convert 128.10.3.8 to 32-bit binary IP address and store in variable *serverip*
  - Use *udp_register* to register (*serverip*, 37, *TIMELPORT*)
  - Repeat twice or until a successful reply arrives
    - Form a UDP message and call *udp_send* to send it
    - Call *udp_recv* to receive a reply, specifying a timeout
  - Handle the reply or declare failure for *TIMEOUT*
  - Call *udp_release* to release resources

Getutime: Function That Uses UDP (Part 1)

```
/* getutime.c - getutime */

#include <xinu.h>
#include <stdio.h>

/*------------------------------------------------------------------------
 * getutime  -  Obtain time in seconds past Jan 1, 1970, UCT (GMT)
 *------------------------------------------------------------------------
 */
status  getutime(
          uint32  *timvar               /* Location to store the result */
        )
{
        uint32  nnow;                   /* Current time in network fmt  */
        uint32  now;                    /* Current time in xinu format  */
        int32   retval;                 /* Return value from call       */
        uid32   slot;                   /* Slot in UDP table            */
        uint32  serverip;               /* IP address of a time server  */
        char    prompt[2] = "xx";       /* Message to prompt time server*/

        if (Date.dt_bootvalid) {        /* Return time from local info  */
                *timvar = Date.dt_boot + clktime;
                return OK;
        }
```

Getutime: Function That Uses UDP (Part 2)

```
/* Convert time server IP address to binary */

if (dot2ip(TIMESERVER, &serverip) == SYSERR) {
        return SYSERR;
}

/* Contact the time server to get the date and time */

slot = udp_register(serverip, TIMERPORT, TIMELPORT);
if (slot == SYSERR) {
        fprintf(stderr,"getutime: cannot register a udp port %d\n",
                                TIMERPORT);
        return SYSERR;
}

/* Send arbitrary message to prompt time server */

if (getlocalip() == SYSERR) {
        return SYSERR;
}       retval = udp_send(slot, prompt, 2);
if (retval == SYSERR) {
        fprintf(stderr,"getutime: cannot send a udp message %d\n",
                                TIMERPORT);
        udp_release(slot);
        return SYSERR;
}
```

Getutime: Function That Uses UDP (Part 3)

```
        retval = udp_recv(slot, (char *) &nnow, 4, TIMETIMEOUT);
        if ( (retval == SYSERR) || (retval == TIMEOUT) ) {
                udp_release(slot);
                return SYSERR;
        }
        udp_release(slot);
        now = ntim2xtim( ntohl(nnow) );
        Date.dt_boot = now - clktime;
        Date.dt_bootvalid = TRUE;
        *timvar = now;
        return OK;
}
```

The ICMP Interface

- Is conceptually similar to the UDP interface
- An application
  - Converts the target's IP address to a 32-bit binary value
  - Calls *icmp_register* to register the remote address and receive a descriptor to use with remaining calls
  - Generates an ICMP request packet and calls *icmp_send* to send the packet
  - Calls *icmp_recv* to receive a reply, specifying a timeout
  - If a *TIMEOUT* occurs, reports failure; otherwise reports success
  - Handles the reply, if a valid reply was received
  - Calls *icmp_release* to release the registered endpoint

Use Of DHCP At Startup

- When we discuss system initialization, we sill see that a computer uses DHCP at startup to obtain an IP address and related information
  - DHCP is only used once (i.e., it is only run during startup)
  - A DHCP message is sent using UDP (i.e., DHCP uses the UDP interface)
- How can a computer send an Internet packet and receive a reply *before* the computer has an IP address?
- Answer: the computer sends its initial DHCP request to a special IP broadcast address of all ones (255.255.255.255 in dotted decimal)
- The IP code maps the IP broadcast address to an Ethernet broadcast address, and broadcasts the packet on the local network

Delayed use Of DHCP

- An interesting process coordination problem arises with DHCP
  - When starting the operating system, the network processes are not yet running
  - Consequence: the startup code cannot block to wait for a DHCP reply
- Our solution: delay using DHCP until an application tries to use the network
  - Start the network processes during system initialization, but do not attempt to use DHCP
  - When an application calls *getlocalip* to obtain the local IP address, use the application process to send a DHCP request
- In essence, DHCP runs as a side effect of requesting the local IP address

The Process Model
For Network Code

Processes And Network Code

- Unlike many parts of an operating system, network code uses multiple processes
- Various operating systems have experimented with variants
- Examples
  - One process per protocol
  - One process per layer of the protocol stack
  - One process to handle each protocol that retransmits

A Synchronous Interface For Network Devices

- Observe
  - Like many systems, Xinu provides a synchronous device interface
  - A process blocks to wait for input
- Consider how Xinu handles a network device (Ethernet, Wi-Fi, etc.)
  - An entry in the device switch table is created for each network hardware device
  - The interface for each device is synchronous
  - Example: when reading from an Ethernet, a process will block until a packet arrives

A Consequence Of A Synchronous Interface

- Packets may arrive over the network at any time, not just in reply to an outgoing request
  - An application on a remote computer may attempt to contact a local application first
  - An application on a remote computer may use ICMP to ping the local computer
- Consequence: a process must be waiting to read and handle the incoming packets

Xinu's Network Input Process

- To handle asynchronous packet arrivals, Xinu uses a separate *network input process*
- The network input process repeatedly
  - Calls *read* on the ETHER device to wait for the next incoming packet
  - Handles the packet (e.g., if the packet contains UDP, the network input process calls *udp_in*)

ARP and Ping

- Question: how should ARP and ping be handled?
- A reply must be sent when a request arrives from another computer
- Possibilities:
  - Have the network input process send replies
  - Have a separate process for ping and a process for ARP

The Network Process And IP Transmission

- Sending an ARP reply is trivial — the network input process can form a reply and write it to the ETHER device
- Sending an IP packet is complex because an ARP exchange may be needed
  - The sender transmits an ARP request
  - When an ARP reply arrives, the sender fills in needed information and sends the IP packet
- A problem: an Internet packet is being sent in response to an incoming ping request and the network input process blocks to wait for an ARP response, a deadlock will occur because no process will be running to read the ARP reply packet

Solving The Output Problem

- To avoid the problem

The network input process must never call a function that blocks to wait for a reply.

- Our implementation uses a separate IP output process, and arranges for the network input process to deposit outgoing IP packets on a queue for the output process to handle
- The IP output process can block waiting for an ARP reply because the network input process remains running

The IP Output Process

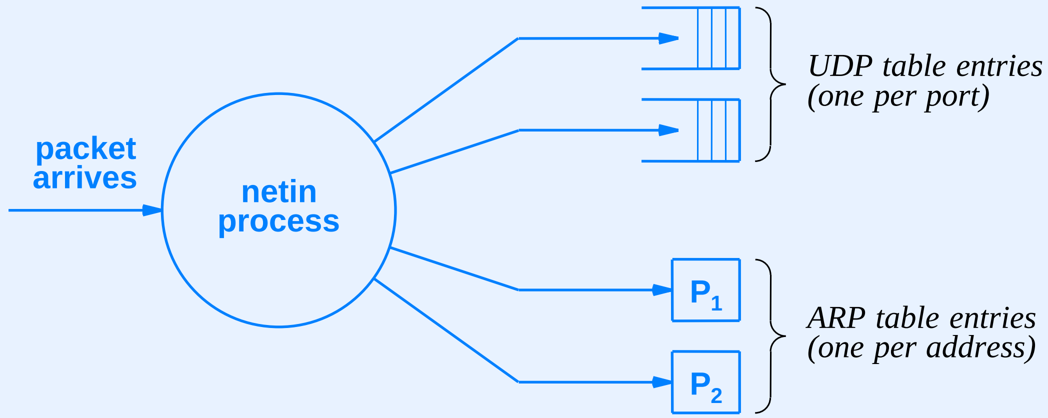
- Runs continuously
- Uses a queue of outgoing packets
- Repeatedly
  - Blocks on a semaphore to wait for a packet in the queue
  - Extracts the next packet
  - Follows the normal output procedure to send the packet, including using ARP
- Note: conceptually, the IP output process isolates the network input process from the output side, allowing them to operate independently

The Resulting Network Process Model

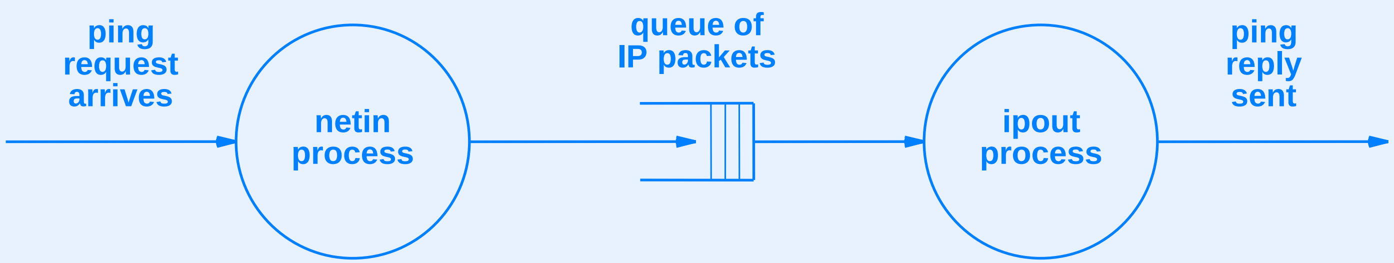
- The Xinu process model is minimalistic (only two network processes)
- The network input process (*netin*)
  - Runs an infinite loop
  - Reads the next Ethernet packet, blocking until a packet arrives
  - Call the appropriate input function to handle the packet (one of *arp_in*, *udp_in*, *icmp_in*)
- The IP output process (*ipout*)
  - Runs an infinite loop
  - Uses a queue of outgoing IP packets
  - Waits until a packet arrives on the queue, and then sends the packet

Illustration Of The Xinu Network Process Model

- *Netin* handles incoming UDP and ARP packets



- *Netin* enqueues ICMP replies for *ipout*, thereby preventing *netin* from blocking



Process Priorities And Incoming Packet Queues

- The *netin* process has a high priority
- An application process may have a low priority
- Consequences
  - An application that is waiting for a packet may not execute immediately after the packet arrives
  - A second packet may arrive for a given application before the first packet has been handled
- To accommodate delayed processing, Xinu uses packet queues to absorb a small burst of packets without discarding any
- Note: the above only applies to UDP and ICMP because ARP packets are processed immediately by the netin process

Implementation Of Packet Queues

- Each UDP or ICMP table entry has a small queue
- If the queue is full when another packet arrives
  - The incoming packet is dropped
  - No error is reported
- Note: dropping packets is possible because UDP and ICMP use best-effort semantics

The ARP Cache And Cache Timeout Processing

- The ARP protocol specifies that the network code must keep a cache of recent address bindings
- Entries in the cache should be removed after 10 minutes
- Is an additional process needed to implement ARP cache timeout?
- The disadvantages of an additional process
  - More context switching overhead
  - Uses system resources, such as stack space, with little real value

The Xinu Approach To Cache Timeout

- To avoid having an extra process handle cache timeout, Xinu uses a trick
- When storing an entry in the cache, Xinu stores the current time in a timestamp field in the entry
- Whenever searching the cache, the code examines the timestamp field in each entry, and removes the entry if the time has expired
- The approach works well for an ARP cache because the cache is only expected to contain a few entries, and the search proceeds sequentially

Summary

- Networking is an essential part of any operating system
- Instead of a completely general purpose implementation of network protocols, Xinu limits the implementation to a few basic Internet protocols and assumes a system only uses an Ethernet network
- Our code uses a single struct that uses unions to specify how protocols are encapsulated
- Only two networks processes are needed in our implementation
  - A network input process (*netin*)
  - An IP output process (*ipout*)
- The general paradigm for our network API is: register an intent to use, send or receive messages, release the entry
- To handle bursts of incoming packets, our code uses queues for incoming packets

Questions?