Computer Networks
ICS 651

- Implementation Strategies
- A simple packet network
- SLIP
- DNS
- homework 2

Implementation Strategies for Packet Networks

- a Sockets API is usually implemented in kernel space and accessed via system calls
- a Sockets API can also be implemented in user space as a library
- Communicating with the network device(s): device drivers, multiple threads, interrupt handlers
- Must be able to deal with more data than we can consume:
  - there might be too much incoming data
  - or too much outgoing data
  - or both
- unreliable protocols can simply discard the excess data, perhaps returning an error code for outgoing data
- reliable protocols must be able to stop the producer of data

Sketch of a very simple implementation

- two threads:
  1. receive thread runs at interrupt time, receives data, delivers data to a data handler
     calling a function in an upper layer, from a lower layer, is an **upcall**
  2. send thread runs at system call time, sends data to the network device (downcall)
- device is the serial line device ("tty"), in "raw" mode, at fixed speed
- this allows transmitting of data bytes
  - usually, also receiving!!! ;)
- how to leverage byte exchange to allow exchanges of packets?
- many possible ways, but to accomodate different-speed devices, would prefer to avoid time-sensitive methods
  - i.e. the end of a packet cannot be defined by a timeout

Bytes to packets: SLIP protocol

- two special characters: END and ESC (escape)
- END marks the end of a packet and the start of the next
- data may contain the special characters:
  - if the data byte is END, we send ESC ENDC
  - if the data byte is ESC, we send ESC ESCC
- where ENDC and ESCC are otherwise normal characters
- predefined maximum user data size, 1006 bytes
- RFC 1055, http://www.ietf.org/rfc/rfc1055.txt
- END is `0xC0`, ESC is `0xDB`, ENDC is `0xDC`, and ESCC is `0xDD`

- This protocol is tolerant of slow communications
- This protocol recovers gracefully from lost bytes

Ways to Convert a Stream of Bytes or Bits to a Stream of Packets

- SLIP inserts additional bytes where needed to preserve the meaning of the special bytes used by the protocol
  - this is called *byte stuffing*
- Could also send the length before each packet
  - what drawbacks would that have?
- Some protocols use a sequence of n 0-bits, or n 1-bits (e.g. n == 5) to mark a frame boundary
- For such protocols, every sequence of n such bits in the data stream must be followed by the opposite bit
  - this is called *bit stuffing*
  - the receiver removes the extra bits
- Such protocols make sense when the underlying physical medium cannot carry too many bits with all the same values
- e.g., if 0-1 and 1-0 transmissions help with synchronization

SLIP Implementation

- `send` system call sends an END character, then repeatedly:
  - either sends the character, or
  - sends the two characters ESC and ESCxxx
- finally, it sends the final END character
- all this is done while holding a lock, so only one thread can send at a time

•data handler has a buffer with 1006 bytes:

- ignore initial END bytes, if any
- put received bytes into buffer, removing ESC sequences
- continue until an unescaped END or we run out of space
- if we found END, deliver the packet, otherwise discard it
- keep state: how many bytes in the buffer, was the last character an ESC, is the buffer in use by a higher layer

SLIP send

- only returns once the packet has been sent

```
    /* this is a macro so the return statement returns from
write_slip_data */
     #define WRITE_BYTE(fd, c)                                  \
         if (write_tty_data (fd, c) != 1) {                     \
           pthread_mutex_unlock (&(send_mutex [fd]));           \
           printf ("slip: error writing tty data\n");           \
           return -1;                                           \
         }
```

SLIP send, part I

```
int write_slip_data (int fd, char * data, int numbytes) {
   int byte;
   if ((numbytes <= 0) || (numbytes > MAX_SLIP_SEND)) {
     printf ("slip: bad size %d\n", numbytes);
     return -1;
   }
 #ifdef DEBUG
   printf ("acquiring send lock for tty %d\n", fd);
 #endif /* DEBUG */
   pthread_mutex_lock (&(send_mutex [fd]));
 #ifdef DEBUG
   print_packet ("sending packet", data, numbytes);
 #endif /* DEBUG */
   /* send the start byte, which of course is called END :-) */
   WRITE_BYTE (fd, END);
```

SLIP send, part II

```
for (byte = 0; byte < numbytes; byte++) {
  unsigned char c = (data [byte]) & 0xff;
  if (c == END) {
    WRITE_BYTE (fd, ESC);
    WRITE_BYTE (fd, ESC_END);
  } else if (c == ESC) {
    WRITE_BYTE (fd, ESC);
    WRITE_BYTE (fd, ESC_ESC);
  } else {                          /* normal byte */
    WRITE_BYTE (fd, c);
  }
}
WRITE_BYTE (fd, END);               /* the end of the frame */
pthread_mutex_unlock (&(send_mutex [fd]));
return numbytes;
}
```

SLIP receive and upcalls

- the receive function is called by the lower layer whenever a new character is received from the device (an *upcall*)
- in turn, once we have a packet, we do the next upcall
- some global variables:

```
static char receive_buffer [MAX_TTYS]
                    [((MAX_SLIP_SIZE + 7) / 8) * 8];
static int receive_position [MAX_TTYS];
static int escaped [MAX_TTYS];
static int error_frame [MAX_TTYS];
/* the data handlers are also global. */
typedef void (* my_data_handler) (int, char *, int);
static my_data_handler slip_data_handler [MAX_TTYS];
```

SLIP receive (without error handling)

```
static void data_handler_for_tty (int tty, unsigned char c)
{
  pthread_mutex_lock (&(receive_mutex [tty]));
  if (escaped [tty]) {    /* last character was an escape */
    escaped [tty] = 0;
    if (c == ESC_END)
      put_char_in_buffer (tty, END);
    else /* c should be ESC_ESC */
      put_char_in_buffer (tty, ESC);
  } else {              /* last character was not ESC */
    if (c == END) {    /* done, give packet to data handler. */
      if (receive_position [tty] > 0)   /* packet is not empty */
        slip_data_handler [tty] (tty, receive_buffer [tty],
                                 receive_position [tty]);
      /* get ready to start receiving a new packet */
      receive_position [tty] = 0;
    } else if (c == ESC) { /* escape for the next character */
      escaped [tty] = 1;
    } else {                /* 'normal' character */
      put_char_in_buffer (tty, c);
    }
  }
  /* make the buffer available to other threads. */
  pthread_mutex_unlock (&(receive_mutex [tty]));
}
```

SLIP receive error handling

- what should we do if we run out of space in the buffer?
- … if an ESC character is followed by something other than ESC_ESC or ESC_END?
- if there is no data handler?
- if the current frame has an error, yet this character is not END?

See http://www2.hawaii.edu/~esb/2016spring.ics651/jan19.html , "SLIP receive"

A Simple Network

- a network that can transfer packets between exactly two computers connected by a serial line, with implicit addressing
- maximum packet size is 1006 bytes
- packets could be lost or corrupted if there are any errors (serial lines are noisy) -- and we do not detect this
- packets are never reordered or duplicated by the network. Could they be reordered by the multithreading?
- If there are many threads sending, the maximum delay could be very large, but if there are only two threads (one send, one receive), the maximum delay is bounded
- software is multithreaded, requires synchronization for the use of shared resources
- SLIP provides a data handler to read bytes: whoever uses SLIP must in turn provide another data handler (a SLIP packet handler)

What we haven’t built (yet)

- more than two computers
- network addresses
- addresses humans can remember
- reliable transmission
- wide-area transmission
- transmission of arbitrary amounts of data

Naming and Addressing

- the first step in expanding this network is to allow for more than two computers
- this means each data packet needs a destination address, since the SLIP "header" does not include such information
- an address identifies a destination
- only one destination should have a given address
- one destination can have one or more addresses
- addresses are usually fixed-size binary numbers and are used by computers -- people prefer meaningful strings, that is, names
- if we use names, we will need a mechanism for converting names to addresses: *name resolution*

Addresses in an Internet

- since computers can only communicate in a point-to-point fashion (so far, in our serial-line network), many computers will have multiple interfaces, so they can forward data from one to the other
- The Internet Protocol (IP) adds a header to each packet, listing the destination IP address
    - the Internet Protocol also specifies how packets are forwarded by computers with multiple interfaces
- in the Internet Protocol, addresses are assigned to network interfaces, not to computers: a computer may have multiple interfaces, each with one (or possibly more) addresses

Addresses in an Internet II

- a large internet is made up of smaller networks, and so benefits from hierarchical addresses
- with a hierarchical address, a packet is first routed to a computer somewhere on the destination network, then to a computer on the destination sub-network, and finally to the destination
- it follows that hierarchical addresses must be assigned depending on the network to which the hardware is connected -- the hardware cannot be preconfigured with the address
- although hierarchies might have multiple levels, the original IPv4 only had two levels: network and host
- fixed-size addresses are processed more efficiently: 32 bits (IPv4), 128 bits (IPv6)

Names in an Internet

- names are useful if they can be translated into addresses
- if arbitrary translations are desired, we use a database (a table)
- a hierarchy is useful for:
    - delegating assignment of names, e.g. to national registration authorities, registrars, registrants, and whoever they delegate to
    - allowing the expression of natural hierarchies, e.g. governments, companies, and all their subsidiary organizations
- the Domain Name System provides such names, with a distributed database
- the process of converting Domain Names to IP addresses is *Domain Name Resolution*, implemented by `gethostbyname` or `getaddrinfo` in Unix systems

Domain Names

- name hierarchy: rightmost "label" is the one nearest the root (the root is simply "."). Example, ".edu." (full name)
- each label is 1-63 characters:
  - starting with a letter
  - containing letters, digits, or hyphens
  - ending with a letter or a digit
- uppercase and lowercase are treated as if they were the same that is, WWW.HAWAII.EDU is the same name as www.hawaii.edu
- each IP address may have any number (0, 1, or more) of names associated with it

Domain Name Zone

- a collection of names that are
  - contiguous in the tree, and
  - administered as one unit,
- is one *zone*
- a zone can be split into two by assigning a subtree to another administration
- a zone should have at least two name servers responsible for providing name-to-address resolution

Domain Name Database

- Domain Name resolution is implemented as a lookup in a distributed database
- lookup is the only distributed operation, so the database is read-only
- each zone has at least one authoritative and one secondary server
- the servers for each zone must be configured with the IP addresses of
    - the servers for the zones above them in the tree, and
    - the servers for the zones below them in the tree (if any)
- a query directed to any server can be referred to a different server, "closer" in the tree to the destination
- a host need only be configured with the IP of a single DNS server, though multiple such addresses are common

Domain Name Protocol

- TCP or UDP (in case of TCP, a two-byte length field is added -- why?)
- a Query requests one resource record
- a Response returns one resource record, if available
- some resource record types:
  - [**A**] name to (IPv4) Address translation
  - [**AAAA**] name to IPv6 address translation
  - [**CNAME**] Canonical NAME for a given DNS name
  - [**MX**] host willing to do eMail eXchange for the given domain name
  - [**NS**] name server authoritative for the domain
- resource records carry a TTL (time-to-live) field, in seconds, e.g. 3600

Domain Name Example

```
 -> dig mx hawaii.edu

 ; <<>> DiG 9.5.1-P2 <<>> mx hawaii.edu
 ;; global options:  printcmd
 ;; Got answer:
 ;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35571
 ;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3,
ADDITIONAL: 4

 ;; QUESTION SECTION:
 ;hawaii.edu.                    IN      MX

 ;; ANSWER SECTION:
 hawaii.edu.             1800    IN      MX      10
mx1.hawaii.edu.

 ;; AUTHORITY SECTION:
 hawaii.edu.             1800    IN      NS      dns2.hawaii.edu.
 hawaii.edu.             1800    IN      NS      dns4.hawaii.edu.
 hawaii.edu.             1800    IN      NS      dns1.hawaii.edu.
```

Domain Name Example II

```
;; ADDITIONAL SECTION:
mx1.hawaii.edu.         1800    IN      A       128.171.224.25
dns1.hawaii.edu.        1800    IN      A       128.171.3.13
dns2.hawaii.edu.        1800    IN      A       128.171.1.1
dns4.hawaii.edu.        1800    IN      A       130.253.102.4

;; Query time: 2 msec
;; SERVER: 128.171.3.13#53(128.171.3.13)
;; WHEN: Mon Aug 31 10:10:00 2009
;; MSG SIZE  rcvd: 169
```

Typical Domain Name Query

- a host sends a request for an A or AAAA resource record to its name server
- the name server may have the resource record. If not, it may query another server, or return the address of another server (in an NS record)
- if no response, query may be resent (server is stateless)
- the search continues until an A/AAAA record is found, or a negative response is received (or until a timeout)
- servers and hosts can cache the resource records up to TTL seconds
- all this for `gethostbyname` or `getaddrinfo` (system calls), or `nslookup`, or `dig`, or `host` (commands)

What does DNS need from its lower layers?

- a network with multiple hosts
- any-to-any communication of packets
- reliability is not required: DNS retransmits the query if it does not get a response (since the database is read-only, queries are idempotent)
- routing based only on IP addresses
- initial configuration:
  - a machine needs to be configured with the address of a DNS server
  - an authoritative DNS server needs the IP addresses of DNS servers of neighboring zones