CMSC417 Spring 2016 Lecture #2 2/1/2016

Agenda

⇒ Review layering (OSI model, class design)
- wires (encoding bits, sharing)
- scale ⇒ rings (actually should be shared wires, but sharing is harder then)
  - □ efficiency of TDMA with many senders is bad
  - □ need O(n) wires to have send/recv wire for each computer
- robustness at scale
  - □ first ⇒ two rings so no single failure kills you
  - □ second ⇒ many interconnected rings so they can fail independently
- independent rings
  - □ how to join them? have special computers in both (or all 3+) called routers
  - □ routers now have to choose how to forward. how do they know? Answer is routing protocols
- application addresses, i.e., TCP/UDP ports

⇒ Real packet headers (Ethernet, IP, TCP/UDP)
- read/write packet headers using structs in C
- network vs. host byte order

CMSC417 Spring 2016 Lecture #2 2/1/2016

Review of Layering|

⇒ 7 layess of OSI model

⇒ What we talked about last time

we'll cover this later

⇒ Layer 1: Just run a wire

How to encode bits: 0v and 5v, clock synchronization, coding, etc.

⇒ Layer 2: How to connect n computers where $n >> 2$?

⇒ connect them in a ring, listen for when your tag comes up

⇒ In reality we want a source tag and destination tag (really addrs)

⇒ Also, usually done as a single wire with multiple computers attached and/or a broadcast domain

⇒ Layer 3: How to deal with scale when you need to connect nodes to multiple networks?

⇒ need to be able to decide which network to send stuff to when you have options

⇒ need a network label/address

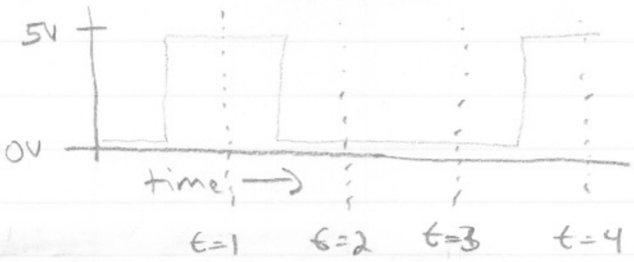
⇒ Layer 4: So far, it's all been about computers, but in reality we care about apps

⇒ add an extra label/addr to pick which app to deliver the packet to

CMSC417 Spring 2016 Lecture #2 2/1/2016

Layering Review Class Design Diagrams)

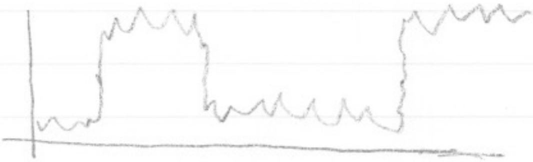
OV vs. 5V digrams (L1 bits on wires)



assume synched clock

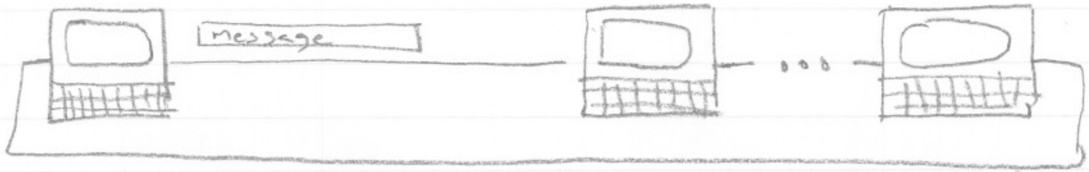
bits 1 0 0 1

actually looks like:



keeping clocks synced is also hard

framing messages (L2 with multiple computers on the wire)



put tag/label/addr for dest CPU at start of message.

⇒ how do we know it's the "start" or "end" of a message?

⇒ how do we know what's "header" vs. "data"

sharing the wire ideas

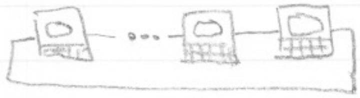
⇒ TDMA, i.e., take turns

⇒ have two different cables, one for sending, one for recieving

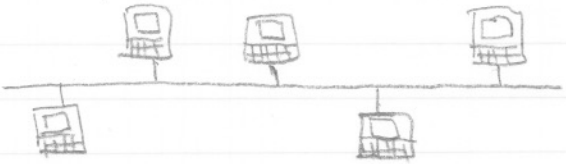
CMSC 417 Spring 2016 Lecture #2 2/1/2016

Reality of Rings vs. Wires

instead of
a ring like this

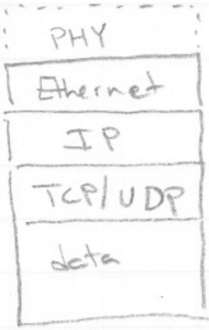


people really just use one shared wire, like this:



sharing is harder, but cabling is easier
⇒ "vampire" taps

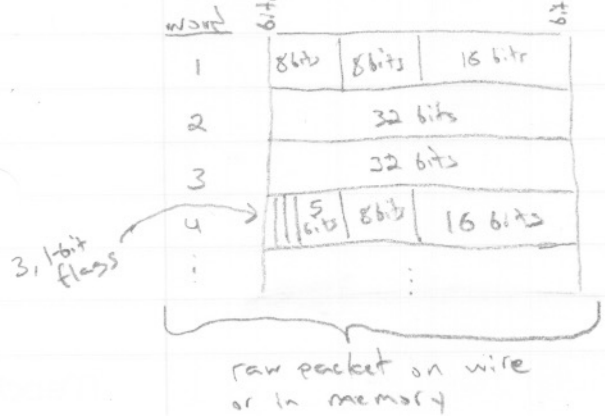
Real headers



} not generally available from s/w b/c
its not representable as "bits"

⇒ can sometimes get info., esp., for WiFi
e.g., bit errors and channel measurements

Structs for headers



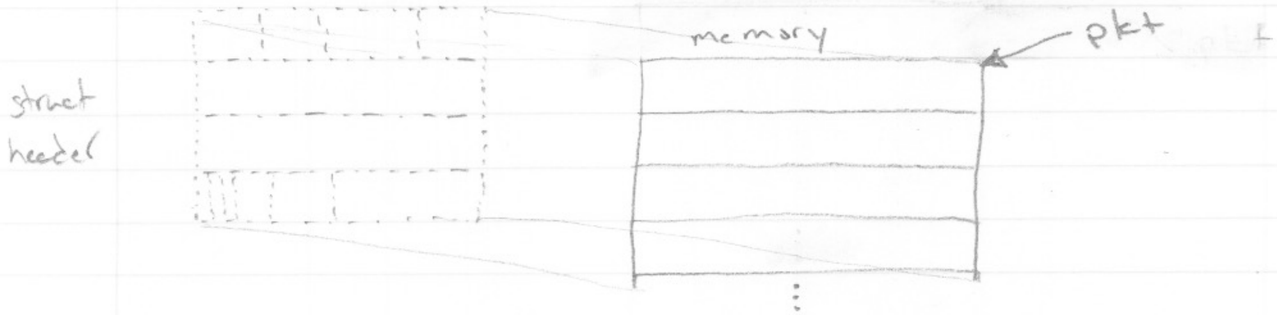
exploit C structs to easily read/write

```
struct header {
    uint8 f1, f2;
   uint16 f3;
   uint32 f4, f5;
   unsigned int f6: 1;
        ⋮
   uint16 f11;
}
```

CMSC417 Spring 2016 Lecture #2 2/1/2016

Structs for headers|

```
char * pkt = <get memory for packet>;
struct header * = pkt;
header -> f1 = <val>;     // assigns vals to fields
header -> f3 = <val>;     // in both header & pkt
   ⋮
```



overlay the struct pointer to act as a "lens" for reading/writing header fields

Byte/bit order|

⇒ different machines lay out data differently
- □ MSB (most significant byte) first
- □ LSB first
- □ Fortunately, bits tend to all be MSb first w/in a byte

⇒ All network data is supposed to be in "Network Byte Order", which is MSB first

⇒ x86, e.g., most every computer you use is LSB first
- □ So, you need to convert
- □ use ntohl, ntohs, htonl, htons to convert shorts and longs in the given direction

⇒ good practice: ① all locals are in host byte order
② all fields in a packet are in ntwk byte order
③ convert when moving between

CMSC 417 Spring 2016 Lecture #2 2/1/2016

select() / poll()

⇒ most network applications need to maintain multiple connections, e.g., fetch content from different web servers

⇒ need to simuntaneously service all the connections

- □ two options : one thread per connection
  make non-blocking calls
- □ send() / recv() can block
- □ select() lets you wait on a number of different sockets and specify a timeout
- □ lots of details that are annoying to get right
  - • fdsets, nfds is max(fds)+1, sets are cleared and used to tell you which sockets are ready, etc.
- □ poll() is newer and provides an interface that some people find easier to use
- □ for real performance at scale use libev and libevent which wrap select() / poll() to avoid linear scans if possible and other tweaks