6.033 Computer System Engineering
Spring 2009

6.033 Lecture 3: Naming

Systems

- system = a bunch of resources, glued together with names
  - often the best way to view or build
  - often most interesting action is in the names
- [cloud with components & connections]
- what are the resources?
  - often can view them as just three basic types

* Fundamental abstractions

- storage
  - mem, disk
  - layered: data struct, FS, disk arrays
- interpreters
  - cpu
  - programming language e.g. java VM
- communication
  - wire, ethernet
  - layered: internet, ssh, unix pipe
- we'll talk about all these abstractions during the course

Why these abstractions?

- many components fit these molds
- they work well together
- hardware versions are cheap
- for these abstractions, general techniques known
  - to increase performance, fault tolerance, functionality

Names

- all three abstractions rely on names
  - memory stores named objects
  - interpreter manipulates named objects
  - communication link connects named entities
- much of sys design is naming --- glue

DNS Example

- what does web.mit.edu mean?
- you can look it up and turn it into an IP address
  - 18.7.22.69
  - Internet knows how to send data to IP addresses
- how does the translation work?

The big picture

- my laptop O/S shipped with IP addrs of a few "root name servers"
  - these are hard-wired, in a file, about a dozen
  - 198.41.0.4 (virginia)
  - 128.63.2.53 (maryland)
  - ...
  - this list doesn't change much, otherwise old computers would break
- my laptop picks a root server, sends it a query,
  - asking for IP address of the string "web.mit.edu"
- each root server has exactly the same table, the root "zone"
  - edu NS 192.5.6.30
  - edu NS 192.12.94.30
  - com NS 192.48.79.30

```
    ...
  finds entries in root zone that matches suffix of "web.mit.edu"
    returns those records to my laptop
  now my laptop knows who to ask about edu
  re-sends request to one of the edu servers
  each edu server has a copy of the edu zone
    nyu.edu NS 128.122.253.83
    mit.edu NS 18.72.0.3
    mit.edu NS 18.70.0.160
  returns the two MIT records to my laptop
  now my laptop forwards request to an MIT server
  each MIT server has a copy of the mit.edu zone
    web.mit.edu A 18.7.22.69
    bitsy.mit.edu A 18.72.0.3
    csail.mit.edu NS 18.24.0.120
    ...

DNS has a hierarchical name space
  diagram of a tree

Where do the zone files come from?
  each zone centrally but independently administered
  root by IANA, an international non-profit org
  com by VeriSign, a company
  MIT IS maintains its zone files, updates them on its servers

Why this design?
  everyone sees same name space (at least for full names)
  scalable in performance
    via simplicity, root servers can answer many requests
    via caching, not really described here
    via delegation (keeps table size down)
    contrast to single central server
  scalable in management
    MIT control own names by running its own servers
  fault tolerant

There are problems:
  who should control root zone? .com?
  load on root servers
  denial-of-service attacks
  security problems: how does client know data is correct?
  how does verisign know i'm ibm.com when I ask it to change NS?

DNS has been very successful
  25 years old
  design still works after 1000x scaling
  you can learn more about DNS from Chapter 4 of the notes

How do systems use DNS?
  user-friendly naming (type a URL)
  web page tells browser what to do on button press
    i.e. connects one page to the next
  Adobe PDF Reader contains DNS name for where (later) to check for upgrades
  allows services to change IP address of servers (change hosting service)
  DNS servers can give different answers to different clients
```

turn www.yahoo.com into IP address near client

Naming goals
  let's step back
  1. user friendly
  2. sharing
    diagram: A and B both know N, refers to ...
  3. retrieval (sharing across time)
    A knows N, refers twice to ... over time
  4. indirection
    diagram: A knows fixed N, refers to variable N', refers to ...
  5. hiding
    hide implementation
    control access (access only if you know name, or hook to hand ACLs)

Naming systems are common, since so powerful
  phone numbers
  files / directories
  e-mail addresses
  function/variable names in programs

Example: virtual addressing
  often called "virtual memory" or "paging"

Why this example?
  naming mechanism
  well designed
  solves many problems
  basis for many neat o/s features

Memory
  this is review from 6.004
  also Chapter 5 in notes
  diagram: CPU, address/data wires, DRAM
  DRAM an array of bytes
  indexed with "physical address"
  let us say 32-bit addresses
  could have up to 4 GB of DRAM
  example: LD R4, 0x2020

Why physical addressing not sufficient?
  many reasons, as we will see
  here's one: what if program too big to fit in DRAM?
    but still fits in 2^32 bytes
    e.g. two-megabyte program, only one megabyte of DRAM

Demand Paging
  diagram: cpu, dram, disk
  want some data to be in memory, some on disk
    when program refers to data that's not in memory,
    move something else from memory to disk,
    load in the desired memory from disk
    called "demand paging"
  that's the original motivation, not so relevant any more

Virtual Addressing Idea

```
  diagram: MMU, two arrays of memory cells
  set of arrows from left to right
  "virtual addresses" vs "physical addresses"
    s/w ONLY loads/stores using virtual addresses
    phys addrs ONLY show up in the mapping table
  conceptually, translations for all 2^32
    some refer to physical memory
    others to disk locations
  how to implement?
    if per-byte mappings, 16 GB!
    not practical to have a translation table with entry per byte addr

Page Tables
  how to make the mapping table small?
  divide virtual addresses into contiguous aligned "pages"
  upper (say) 20 bits are "virtual page number"
  MMU only maps page number -> upper 20 bits of a phys address
    then use lower 12 va bits as lower 12 pa bits

Example:
  page table:
    0: 3 (i.e. phys page 3)
    1: 0
    2: 4
    3: 5
  register R3 contains 0x2020
  LD R4, (R3)
  CPU sends (R3)=0x2020 to MMU
    virtual page number is 2
    offset is 0x20
    thus pa = 4*4096 + 0x20 = 0x4020
  MMU sends 0x4020 on wires to DRAM system

Page table small enough to fit in DRAM
  has 2^20 entries, probably 4 bytes each
  now page table is only 4 MB, not so big

Where is the page table?
  in memory
    so can be modified with ordinary instructions
  CPU Page Table Base Register points to base
  so a memory reference conceptually requires *two* phys mem fetches
    one to get the page table entry
    one to get the actual data
  allows us to keep multiple page tables, quickly switch
  will come in handy later...

Flags
  "Page Table Entry"
  20 bits, plus some flags
  Valid/Invalid, Writeable/ReadOnly
  MMU only does translation if valid and correct read/write
  otherwise forces a "page fault":
    save state
    transfer to known "handler" function in operating system
```

```
Demand paging
  page table:
    0: 1, V
    1: -, I
    2: 0, V
  and only two pages of physical memory!
  if program uses va 0x0xxx or 0x2xxx, MMU translates
  fault if program uses va 0x1xxx
  handler(vpn):
    (ppn, vpn') = pick a resident page to evict
    write dram[ppn] -> disk[vpn']
    pagetable[vpn'] = -, I
    read dram[ppn] <- disk[vpn]
    pagetable[vpn] = ppn, V

Demand paging discussion
  picking the page to evict requires cleverness
  e.g. least recently used
  we are treating DRAM as a cache for "real" memory on disk
  can do that b/c programs use "names" (va) for memory
  we have full control over what the names mean
    map to DRAM
    or arbitrary action in page fault handler

What do operating systems use page tables for?
  Demand paging
  Lazy loading of big programs
  Zero fill
  Address spaces for modularity, defend against bugs
    one page table per running program
    one program cannot even name another program's memory!
  Shared memory for fast communication
    two different names for same physical page
    tight control over what's shared
  Copy-on-write fork
  Distributed shared memory

Closing
  keep your eyes out for naming systems
  be aware in your own work
    when introducing naming might help

****

finish up at start of next day

using virtual addressing as an example of naming
  has turned out to be powerful, many uses of one underlying mechanism
diagram: cpu, mmu, ram, simplified mapping table
programs "name" data with virtual addresses
mapping is valid (refers to dram), or causes "page fault" to o/s
  o/s fills in entry, returns
  program never the wiser
what can you do with this naming mechanism?
  demand page from disk: support big programs
  lazy start from disk
```

address spaces: different mappings for different programs
  o/s switches depending on which program is currently running
  cannot even *name* each other's memory
  contains bugs
shared memory, e.g. X client and server (on same machine)
distributed shared memory
  build parallel programs on a cluster of separate machines
    but with shared memory, all one big program
  diagram: draw other machine, its ram and map
  page fault: fetch, mark him invalid

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 4: Client/server

Modularity
  how to impose order on complex programs?
  break into modules
  [big Therac-25 blob => components, interactions]
  goal: decouple, narrow set of interactions

Modularity (2)
  what form does modularity take?
  interactions are procedure calls
    C -> P, P returns to C
  procedure clarifies interface, hides implementation
    P(a) { ... }
    C() { ... y = P(x); ... }

Enforced?
  is the interface enforced?
  is the implementation hidden?

What actually happens when C calls P?
  6.004
  they communicate via a shared stack, in memory
  [stack: regs, args, RA, P's vars, maybe args...]
  C: push regs, push args, push PC+1, jmp P, pop args, pop regs, ... R0
  P: push vars, ..., pop vars, mov xx -> R0, pop PC

Call Contract
  P returns
  P returns to where C said
  P restores stack pointer
  P doesn't modify stack (or C's other memory)
  P doesn't wedge machine (e.g. use up all heap mem)

Soft modularity
  at a low level, none of call contract is enforced!
  want: spec + contract
  spec: we cannot hope to enforce (e.g. does sqrt() return the right answer?)
  contract: is purely mechanical, we can try to enforce
  goal: programmer need only worry about spec, not contract
  == Enforced Modularity
    there are many ways to enforce modularity
    we'll look at one today

Client/server
  note much of prob from sharing same machine and memory
  so: put C and P on separate machines
  interact only w/ msgs
  [diagram: two boxes with a wire]
  Examples: web client / server, AFS, X windows (about to read)

Code + time diagram
  Client:
    put args in msg (e.g. URL)
    send msg
    wait for reply

```
    read result from reply
  Server:
    wait for req msg
    get args
    compute...
    put result in reply msg (e.g. content of index.html) send reply
    goto start

C/S Benefits
  1. protects control flow (e.g. return address)
  2. protects private data
  3. no fate sharing (crashes and wedges don't propagate)
  4. forces a narrow spec (for better or worse, no global vars)
  c/s enforces most of call contract
    bugs can still propagate via messages
    but that's a fairly restricted channel
    easier to examine than all of memory
  spec still programmer's problem: ensure server returns the right answer

C/S helps with Multiple Parties
  it turns out c/s has benefits more than just enforced modularity
  [diagram: emacs, AFS, print]
  emacs on workstation, AFS, print server
  workstation uses multiple servers
  AFS has multiple clients
  print server might be both server and client of AFS
    e.g. client sends file name to print server

AFS server is a "Trusted Intermediary"
  1. get at your data from multiple physical locations
  2. share with other clients
  3. control the sharing (private vs public data, ACLs)
     private, friends, public
     bugs in one client don't affect other clients
  4. servers physically secure, reliable, easy to find
  these c/s benefits are as important as enforcing modularity
    e.g. e-mail server, eBay

RPC
  c/s involves a lot of nasty error-prone msg formatting &c
  why not hide details behind real procedure interface?

Example:
  imagine this is your original single-program ebay implementation:
  ui(){
    print bid("123", 10.00);
  }
  bid(item, amt) {
    ...
    return winning;
  }
  you want to split at bid() w/o re-writing this code

Client stub
  idea: procedure that *looks* like bid() but sends msg
  client stub:
```

```
    bid(item, amt){
      put item and amt in msg;
      send msg;
      wait for reply;
      winning <- reply msg;
      return winning;
  now original ui() is using a server!

Server stub
  want to use original bid() code on server
  server stub:
  dispatch(){
    while(1){
      read msg
      item, amt <- msg;
      w = bid(item, amt)
      msg <- w;
      send msg;
    }
  }

Marshal / unmarshal
  need standard way to put data types into messages
    a message is a sequence of bytes
  standard size for integers, flat layout for strings and arrays
  typical request message format:
    proc# id 3 '1' '2' '3' 10.00???

Automatic stub generation
  tool to look at argument and return types: run it on bid()
  generate marshal and unmarshal code
  generate stub procedures
  saves programming
  ensures agreement on e.g. arg types

RPC very successful at simplifying c/s, but

RPC != PC
  despite syntactic similarity
  amazon web, warehouse back-end
  [time-line]
  user (browser) wants to place an order
  RPC 1: check inventory(isbn) -> count
    what if network loses/corrupts request?
    loses reply?
    warehouse crashes?
    stub can hide these failures by timeout + resend
      wrap loop around stub
    leads to duplicates -- OK, no side effects
  RPC 2: ship(isbn, addr) -> yes/no
    request lost?
    reply lost?
    warehouse crashes?
    can stub hide these failures by retrying?
    are duplicates OK?
```

How do RPC systems handle failure?
  a couple of approaches, often called "failure semantics"
  1. at least once
     client keeps re-sending until server responds
     -> more than once
     -> error return AND success
     only OK for RPCs w/o side effects e.g. check inventory
  2. at most once
     client keeps re-sending (may time out + error)
     server remembers requests and suppress duplicates
     -> error return AND success
     is it OK for ship()?
       not if we also want to charge credit card iff book shipped
       if client gives up, or crashes, did the book ship?
  3. exactly once
     this is often what we really want: just do it
     hard to do in a useful way, will see practical versions later
  most RPC systems do #1 or #2

RMI code slide
  you feed first part to the rmic stub generator
    it produces implementation of BidInterface for main() to call
    and server dispatch stubs
    the server stubs call your implementation of bid() (not shown)
  public interface ensures client/server agree on arguments (msg format)
  procedure call to bid() looks very ordinary!
    but around it there are non-standard aspects
  Naming.lookup() -- which server to talk to?
    server has registered itself with Naming system
    lookup() returns a proxy BidInterface object
    has bid() method, also remembers server identity
  try/catch
    this is new, every remote call must have this, for timeouts/no reply

Summary
  Enforced modularity via C/S
  RPC automates details
  RPC semantics are different
  Next: enforced mod on same machine

RMI code slide (l4.ppt):

```
public interface BidInterface extends Remote {
  public String bid(String) throws RemoteException;
}

public static void main (String[] argv) {
    try {
      BidInterface srvr = (BidInterface)
           Naming.lookup("//xxx.ebay.com/Bid");
      winning = srvr.bid("123");
      System.out.println(winning);
    } catch (Exception e) {
      System.out.println ("BidClient exception: " + e);
    }
  }
```

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 5: Operating System Organization

Plan for next few lectures
- general topic: modularity
- just talked about one module per computer (c/s)
  - more details on net-based c/s later in course
- now: modules within one computer
  - L5: o/s organization -- tools for enforced modularity
  - L6: concurrency
  - L7: threads
- o/s interesting design artifact in itself

O/S Goals
- multiplexing: one computer, many programs
- cooperation: help programs interact / communicate / share data
- protection (bugs, privacy)
- portability (hide h/w differences and e.g. amt of RAM)
- performance (improve, and don't get in the way)

Key techniques to achieve those goals?
- virtualization
- abstraction

Virtualization
- computer has h/w resources
- a few CPUs, one mem system, one disk, one net i/f, one display
- but you want to run many programs (X, editor, compiler, browser, &c)
- idea: give each program a set of "virtual" resources
  - as if it had its own computer
  - CPU, memory

Virtualization example: CPU
- multiplex one CPU among many programs
- so each program thinks it has a dedicated CPU
- give the CPU to each in turn for 1/n-th of the time
- h/w timer that interrupts 100 times/second
- save registers of current program (CPU registers)
- load previously-saved registers of another program
- continue
- transparent to the programs!
- much more to say about this in the next few lectures

some systems virtualize all h/w
- "virtual machine monitors"
- give each subsystem (program?) a complete virtual computer
  - precisely mimic complete hardware interface
- early IBM operating systems shared computer this way
  - one computer, many VMs
  - each user can run their own o/s
- VMWare, Parallels on Mac, Microsoft Virtual PC
- virtualization by itself makes cooperation hard
  - compiler wants to see editor's output
  - not so easy if each thinks it has a dedicated virtual disk

Abstraction
- abstraction: virtualize but change interface

```
  change interfaces to allow sharing, cooperation, portability

Abstraction examples:
  really "abstract virtual resources"
  disk -> file system
  display -> window system
  cpu -> only "safe" instrs, e.g. no access to MMU, clock, &c
  ??? -> pipes

What does an abstraction look like?
  usually presented to programmer as set of "system calls"
  see slide (l5.ppt)
  you're familiar with FS from UNIX paper
  chdir asks the kernel to change dir
  rest of code reads a file
  system call looks like procedure, we'll see it's different internally

main() {
  int fd, n;
  char buf[512];

  chdir("/usr/rtm");

  fd = open("quiz.txt", 0);
  n = read(fd, buf, 512);
  write(1, buf, n);
  close(fd);
}

How to manage / implement abstractions?
  how to enforce modularity?
    e.g. want prog to use FS *only* via system calls
  mechanism: kernel vs user
  [diagram: kernel, programs, disk, &c]
  kernel can do anything it likes to h/w
  user program can *only* directly use own memory, nothing else
    asks kernel to do things via system calls
  w/ this arrangement, enforcement more clear, two pieces:
    keep programs inside their own address spaces
    control transfers between user and kernel

enforcing address space boundaries
  use per-program pagemap to prohibit writes outside address space
    kernel switches pagemaps as it switches programs
    that's how o/s protects one program from another
  program's address space split
    kernel in high addresses
    user in low addresses
  PTEs have two sets of access flags: KR, KW, UR, UW
    all pages have KR, KW
    only user pages have UR, UW
    convenient: lets kernel directly use user memory, to impl sys calls

CPU has a "supervisor mode" flag
  if on, Kx PTE flags apply, modify MMU, talk to devices
  if off, Ux PTE flags, no devices
```

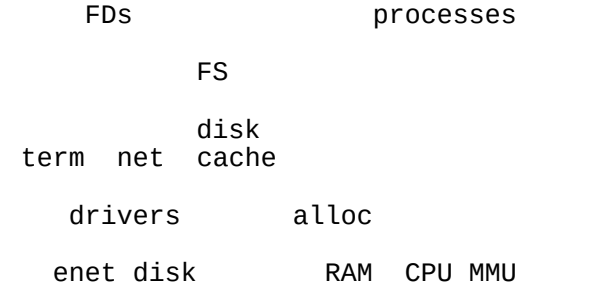
```
  need to ensure supervisor mode = on only when exec kernel code

orderly transfer from user to kernel
  1. set supervisor flag
  2. switch address spaces
  3. jmp into kernel
  and prevent user from jumping just anywhere in kernel
  and can't trust/user user stack
  can't trust anything user provides, e.g. syscall arguments

system call mechanics
  App:
    chdir("/usr/rtm")
      R0 <- 12 (syscall number)
      R1 <- addr of "/usr/rtm"
      SYSCALL
  SYSCALL instruction:
    save user program state (PC, SP)
    set supervisor mode flag
    jump to kernel syscall handler (fixed location)
  Kernel syscall handler:
    save user registers in per-program table
    set SP to kernel stack
    call sys_chdir(), an ordinary C function
    ...
    restore user registers
    SYSRET
  SYSRET instruction:
    clear supervisor mode flag
    restore user PC, SP
    continue process execution

this is procedure call with enforced modularity
  though only protects kernel from user program
  user program can do only two things
    use just its own memory
    or jump into kernel, but then kernel is in charge
  SYSCALL combines setting of supervisor mode and jumping to known location
    would be a disaster to let user program specify target

how to use kernel's enforced modularity?
  how to arrange module boundaries for system services?
  two main camps: microkernel vs monolithic

the microkernel vision
  implement main o/s abstractions as user-space servers
    FS, net, windows/graphics
  apps, servers interact with messages
  kernel provides minimum to support servers
    inter-process communication (messages / RPC)
    processes -- address space + running program
  most interaction via messages
    only two main system calls: send() and recv()

why are microkernels attractive?
  elegant, simple
```

  direct support for client/server
  kernel is small, fewer bugs
  kernel can be optimized to do one thing well (messages)
  hard modularity among different servers, vs bugs
  uniform port scheme -> uniform security plan
    control access by giving (or not giving) port access rights
  easier to distribute (messages allows services on other hosts)

where did this idea go?
  implementations were slow around 1990 -- e.g. Mach
    many messages, esp if you have many servers
  tended to end up with a few huge servers, so little modularity win
  the message idea was influential, e.g. Apple's OSX

monolithic kernels -- the (so far) winning design
  it's too much of a pain to separate lots of servers
  easier and more efficient to have one big program

what's in the Linux kernel?



complex: 300+ sys calls, not just two+ as in microkernel
  much of complexity hidden beneath uniform interfaces -- OO style
  e.g. all storage devices look the same to the file system
    USB flash key, hard disk
  all network hardware looks the same to network code
  so not as complex as it looks

why have monolithic kernels mostly won vs microkernels?
  tradition, advantages of microkernels are subtle
  efficiency: procedure call or mem refs rather than messages
  efficiency: better algorithms via integration among modules
    balance between file system cache and amt of mem for processes
  easy to add microkernel-style messaging, user-level servers
    X, sshd, DNS, apache, database, printer

what doesn't work so well in [monolithic?] kernels?
  device drivers are buggy, cause crashes
  lots of rules e.g. for user pointers passed as arguments
    tricky bug-prone programming environment
  bad at shared resource management (mem, disk)
    compiler uses lots of mem -> laptop unuseable
    backup program uses disk -> laptop unuseable (so I don't back up...)
    modularity is soft for resource management

Summary

o/s virtualizes for sharing/multiplexing
abstracts for portability and cooperation
provides enforced modularity in two useful ways:
  program / kernel
  program / program
supports two kinds of c/s: program -> kernel, program -> program
next: specific techniques for client/server on single computer

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 6: Client/server on one computer

Intro
  how to implement c/s on one computer
  valuable in itself
  involves concurrency, an independently interesting topic
    DP1 is all about concurrency

what do we want?
  [diagram: X client, X server, NO KERNEL YET]
  client wants to send e.g. image to server
  goal: arms-length, so X srvr not vulnerable to client bugs

idea: let kernel manage interaction
  client/srvr interact w/ trusted kernel, not each other
  [diagram: X client, X server, kernel]
  let's focus on one-way flow (can use two for RPC)
  buffer memory in kernel
    each entry holds a message pointer
  send(m) to add msg to buffer
  receive(m) to read msg out of buffer
  finite buffer, so send() may have to wait
  buffer may be empty, so receive() may have to wait
  why does buffer have multiple entries?
    sender / receiver rates may vary around an average
    let sender accumulate a backlog when it's faster
      so receiver has input when sender is slower
  very much like a UNIX pipe

problem: concurrency
  some data structure inside kernel, s() and r() modify it
  what if s() and r() active at same time? may interfere
  concurrency a giant problem, will come up many times
  let's start simple:
    each program gets its own CPU
    there is only one memory system
  [diagram: two CPUs, one memory]
  system calls run on both CPUs!
    i.e. if program A calls send(), send() runs on A's CPU
  send() and receive() interact via single shared memory system

data structure
  "bounded buffer"
  [diagram: BUFFER[5], IN, OUT]
  each array entry: pointer to message buffer
  IN: number of messages put into BB
  OUT: number of messages read out of BB
  IN mod 5 is next place for send() to write
  OUT mod 5 is next place for receive() to look
  example: in = 28, out = 26
    two messages waiting, slots 1 and 2
  in > out => BB not empty
  in - out < 5 => not full

send() code slide
  p is "port", points to instance of BB, so we can have many of them

```
    e.g. one per c/s pair, or one per UNIX pipe
  loop to wait until room ("busy-wait")
  write slot
  increment input count

receive() code slide
  loop to wait until more sends than recvs
  if there's a message
  increment p.out AFTER copying msg
    since p.out++ may signal send() to overwrite
    if send() is waiting for space

I believe this simple BB code works
  [show slide with both]
  even if send() and receive() called at same time
  concurrency rarely work out this well!

Assumptions for simple BB send/recv
  1. One sender, one receiver
  2. Each has its own CPU (otherwise loop prevents other from running)
  3. in and out don't overflow
  4. CPUs perform mem reads and writes in the order shown
     oops! this code probably won't work as shown!
     compiler might put in/out in regs, not see other's changes
     CPU might increment p.in before writing buffer[]
     I will assume memory R/W in program order

Suppose we want multiple senders
  e.g. so many clients can send msgs to X server
  would our send() work?

Concurrent send()
  A: send(p, m1) B: send(p, m2)
  what do we *want* to happen?
  what would be the most useful behavior?
  goal:
    two msgs in buf, in == 2
    we don't care about order

Example prob w/ concurrent send()
  on different cpus, at the same time, on the same p
  A          B
  r in, out
             r in, out
  w buf[0]
             w buf[0]
  r in=0
             r in=0
  w in=1
             w in=1
  result: in = 1, one message in bounded buffer, and one was lost!

This kind of bug is called a "race"
  once A puts data in buffer, it has to hurry to finish w/ incr of p.in!

Other races in this code
```

```
  suppose only one slot left
  A and B may both observe in - out < N
  put *two* items in buf, overwrite oldest entry

Races are a serious problem
  easy mistake to make -- send() looks perfectly reasonable!
  hard to find
    depends on timing, may arise infrequently
    e.g. Therac-25, only experienced operator typed fast enough

How to fix send()'s races?
  original code assumed no-one else messing w/ p.in &c
    only one CPU at a time in send()
    == isolated execution
  can we restore that isolation?

Locks
  a lock is a data type with two operations
    acquire(l)
    s1
    s2
    release(l)
  the lock contains state: locked or unlocked
  if you try to acquire a locked lock
    acquire will wait until it's released
  if two acquire()s try to get a lock at same time
    one will succeed, the other will wait

How to fix send() with locking?
  [locking send() slide]
  associate a lock w/ each BB
  acquire before using BB
    release only after done using BB
  high-level view:
    no interleaving of multiple send()s
    only one send() will be executing guts of send()
    likely to be correct if single-sender send() was correct

Does it matter how send() uses the lock?
  move acquire after IF? [slide]

Why separate lock per bounded buffer?
  rather than e.g. all BBs using same lock
    that would allow only one BB to be active
  but it's OK if send()s on different BBs are concurrent
  lock-per-BB improves performance / parallelism

Deadlock
  big program can have thousands of locks, some per module
  once you have more than one lock in your system,
    you have to worry about deadlock
  deadlock: two CPUs each have a lock, each waiting for other to release
  example:
    implementing a file system
    need to ensure two programs don't modify a directory at the same time
    have a lock per directory
```

```
      create(d, name):
        acquire d.lock
        if name exists:
          error
        else
          create dir ent for name
        release d.lock
    what about moving a file from one dir to another? like mv
      move(d1, name, d2):
        acquire d1.lock
        acquire d2.lock
        delete name from d1
        add name to d2
        release d2.lock
        release d1.lock
    what is the problem here?

Avoiding deadlock
  look for all places where multiple locks are held
  make sure, for every place, they are acquired in the same order
    then there can be no locking cycles, and no deadlock
  for move():
    sort directories by i-number
    lock lowere i-number first
    so:
      if d1.inum < d2.inum:
        acquire d1.lock
        acquire d2.lock
      else:
        acquire d2.lock
        acquire d1.lock
  this can be painful: requires global reasoning
    acquire l1
    print("...")
  does print() acquire a lock? could it deadlock w/ l1?
  the good news is that deadlocks are not subtle once they occur

Lock granularity
  how to decide how many locks to have, what they should protect?
  a spectrum, coarse vs fine
  coarse:
    just one lock, or one lock per module
    e.g. one lock for whole file system
    more likely correct
    but CPUs may wait/spin for lock, wasting CPU time
    "serial execution", performance no better than one CPU
  fine:
    split up data into many pieces, each with a separate lock
    different CPUs can use different data, different locks
      operate in parallel, more work gets done
    but harder to get correct
    more thought to be sure ops on different pieces don't interact
      e.g. deadlock when moving between directories
  always start as coarse as possible!
    use multiple locks only if you are forced to, by low parallel performance
```

```
How to implement acquire and release?
  Here's a plan that DOES NOT WORK:
    acquire(l)
      while l == 0
        do nothing
      l = 1
    release(l)
      l = 0
  Has a familiar race:
    A and B both see l = 0
    A and B both set l = 1
    A and B both hold the lock!

If only we could make l==0 test and l=1 indivisible...
  most CPUs provide the indivisible instruction we need!
  differs by CPU, but usually similar to:
    RSM(a)
      r <- mem[a]
      mem[a] <- 1
      return r
  sets memory to 1, returns old value
  RSM = Read and Set Memory

How does h/w make RSM indivisible?
  a simple plan
  two CPUs, a bus, memory
  only one bus, only one CPU can use it
  there's an arbiter that decides
  so CPU grabs bus, does read AND write, releases bus
  arbiter forces one RSM to finish before other can start

How to use RSM for locking?
  acquire(l)
    while RSM(l) == 1
      do nothing
  always sets lock to 1
    if already locked: harmless
    if not locked: locks
  RSM returns 0 iff not already locked
  only one of a set of concurrent CPUs will see 0
  tidbit: you can implement locks w/ ordinary LOADs and STOREs
    i.e. w/o hardware-supported RSM
    it's just awkward and slow
    look up Dekker's Algorithm

Summary
  BB is what you need for client/server on one computer
  Concurrent programming is tough
  Locks can help make concurrency look more sequential
  Watch out for deadlock
  Next: more than one program per CPU
```

6.033 Computer System Engineering
Spring 2009

6.003 Lecture 7: Threads and Condition Variables

topic: virtual processors / threads
  monday: client/server / bounded buffer w/ one CPU per program
  today: more programs than CPUs
    only one CPU (no busy looping!)
    or a few CPUs, many more programs
  also fewer programs than CPUs (CPUs may need to be idle!)
  goal: virtualize the processor
    multiplex CPU among many "threads"

thread abstraction
  state of a runnable program
  so CPU multiplexing == suspend X, resume Y, suspend Y, resume X
  other abstractions for multiplexing CPU are possible
    this is a useful and traditional one
  controlled by "thread manager" or "scheduler"
  what is the required state?
  how to save state for suspend?
  how to resume from saved state?

send() from previous lecture
  illustrates why we want threads and multiplexing
  [send slide]
  loops waiting for BB to be non-full
  burns up a lot of CPU time
  if one CPU, maybe receive will never run!
  we'd like to let receive() run...

send w/ yield
  [send/receive slide]
  yield() gives up the CPU
  lets other threads run
    e.g. a receive() may have been waiting and called yield()
  someday yield() will return
    after other thread yield()s
    e.g. it tries to receive() but BB is empty

how to implement yield()?
  yield() is the guts of the thread implementation
  suspend one, resume another

data:
  threads[] table: state, sp
  thread stack 0, stack 1, ...
  cpus[] table: thread
  t_lock (coarse granularity...)

[yield version 1 slide]

what happens in yield?
  send calls yield
  how does it know what thread is running?
    per-CPU register CPU() contains cpu #
    cpus[] says what's happening on each CPU
  RUNNING -> RUNNABLE

```
    RUNNING means some CPU executing it now
    RUNNABLE means not executing, but could
  save SP (the CPU register)
  look for a different thread to run
    ignore the RUNNING ones
  mark "new" thread as RUNNING, so no other CPU runs it
  restore saved SP of new thread
    that is, load it into CPU's SP register
  return

questions:
  what state does yield save?
    just SP
    what is on the stack? local vars, RA, send()'s saved registers &c
    we might need to save/restore callee-saved registers too
  what happens in return after restoring?
  this use of SP might not work, depends on compiler
    I'm assuming compiled code does not change SP in body of yield()
    and that return basically just pops RA off stack
    more complex in real life
  what does t_lock protect?
    indivisible set of .state and .sp
    indivisible find RUNNABLE and mark it RUNNING
    don't let another CPU grab current stack until we've switched

Questions?

motivate notify / wait
  [send with yield slide]
  send() and receive() still chew up CPU time
    e.g. send() waits for receive to free up a slot in BB
    e.g. receive() waits for BB to be non-empty
    repeated yield() expensive if many threads waiting
  want send to suspend itself
    have receive wake it up when there is space
    do it in a general way
    don't want receive to have to know abt all threads waiting in send

"condition variable"
  object that acts as a rendezvous
  two methods:
    wait(cvar, lock) -- release(lock), yield, return after notify(cvar)
    notify(cvar) -- wake up all threads currently in wait(cvar)
  notify has no memory: if no threads wait()ing, no effect at all
    wait() and then notify(): wait returns
    notify() and then wait(): wait does not return

each BB has two condition variables:
  notfull (send waits on this if full)
  notempty (recv waits on this if empty)

[send with wait/notify]
  if full, waits, receive will someday free up a slot and notify(p.notfull)
  waits in a loop, re-checks after wait returns
    maybe multiple senders waiting, but only one slot freed up
    that is, wait() returning is only a hint
```

```
    you always ought to explicitly check the condition
  notifies notempty in case one or more receives are waiting
    no harm if no-one is waiting
  holds lock across while test and use of buffer
    so no other send() can sneak in and steal buffer[] slot

why does wait() release p.lock? why not have send() release it?
  i.e. why not
    while p.in - p.out == N:
      release
      wait(p.notfull)
      acquire
  notify might occur between release and wait
    no effect, since no threads waiting at that point
    then send()'s wait() won't return, even though there's a msg!
  this is the "lost notify" problem

avoiding lost notifies
  wait(cvar, lock)
    caller must hold the lock
    wait() atomically releases lock and marks thread as waiting
      so no notify can intervene
    re-acquires lock before returning
  notify(cvar)
    caller must hold the lock
  so, implicitly, condition variable always associated with a particular lock

implementing wait
  thread table additions:
    new state: WAITING
    threads[].cvar (so notify can find us)
  big Q: where to release the lock?
  [wait() slide]
  acquire t_lock first, then release the lock, then WAITING
    ensure that notify() holds both!
  b.t.w. need to modify yield()
  [wait+notify() slide]
  notify() caller holds lock, notify() acquires t_lock
    so receive's notify() holds both locks
    either executes before send acquires lock
      or after sending thread suspends
      (but NOT between send's check and suspension)
    if before:
      send() acquire waits until receive is done
      send() will see empty slot and not wait
    if after:
      notify() will see WAITING send thread, and mark it RUNNABLE

but now we must revisit yield()
  [yield v1 again]
  t_lock already held, not need to set state (easy)
  yield might find there is nothing RUNNABLE!!! (harder)
    this thread WAITING, but receive() running on another CPU
  loops forever while holding t_lock
    so no other CPU can execute notify()
    so no thread will ever be RUNNABLE
```

```
    system will hang

how to fix yield()?
  [yield version 2 slide]
  don't acquire t_lock, don't set to RUNNABLE
  release+acquire in "idle loop"
    still spins indefinitely while no runnable threads
    BUT lets other CPUs execute notify()
  t_lock held on return, but wait() releases it
  note I've also set the SP to a per-CPU stack, before idle loop
    why?
    yield() v1 runs idle loop on calling thread's stack
    someone might notify() it
    some other CPU in idle loop might run the thread
    now two CPUs are executing on the same stack
      e.g. calling functions like acquire, which modify the stack
    thus per-CPU stack for yield() to use when not in any thread

pre-emptive scheduling
  what if a thread never calls yield()?
    we are in trouble, no way to multiplex that CPU
    compute-bound, or long code paths, or broken user programs
    too annoying to require programmer to insert yield()s
  we want forcible periodic yield

how to pre-empt?
  timer h/w, generates an interrupt 10 times per second
  interrupt saves state, jumps to handler in kernel
  timer():
    yield()
    return

will the resulting stack resume correctly?
  interrupt pushes PC + regs on current thread's stack
  when not running, stack looks like:
    ...
    RA to thread at time of interrupt
    registers
    RA to timer()
  so yield() returns to interrupt handler, which returns to interrupted code

what if timer interrupt while you are in yield already?
  would call yield recursively
  deadlock: already holding t_lock
  acquire should disable interrupts
    release should re-enable
  not just for here, but all uses of locks

what if timer interrupt after idle loop releases t_lock?
  again, recursive yield()
  but invalid cpus[][CPU()].thread
  so fix yield() to null out .thread
  and fix timer interrupt to yield only if valid .thread

Summary
  closing thought: how to kill a thread? might be running...
```

threads are virtual processors
  allow many threads, few CPUs
  the foundation of time-sharing
we had to integrate:
  yield()
  condition variables
  interrupts for pre-emption
missing: creation (easy), exit (harder)

6.033 Computer System Engineering
Spring 2009

6.033 2009 Lecture 8: Performance

Performance: why am I lecturing about it?
  often has huge effect on design
  often forces major re-design as loads grow
  faster CPUs haven't "solved" performance
    problems have grown too, e.g. Internet-scale
    disk and net latency hasn't improved

Performance Introduction
  your system is too slow
    [diagram: clients, net, server CPU, server disk]
    perhaps too many users, and they are complaining
  what can you do?
  1. measure to find bottleneck
     could be any of the components incl client
     you hope to find a dominating bottleneck!
  2. relax bottleneck
     increase efficiency or add hardware

Decide what you mean by "performance"
  throughput: requests/second (for many users)
  latency: time for a single request
  sometimes inverses:
    if it takes 0.1 second of CPU, and one CPU, then throughput = 10/sec
  often not inverses
    w/ 2 CPUs, latency still 0.1, but throughput 20/sec
    queuing and pipelining

I will focus on throughput, appropriate for heavily loaded systems
  most systems gets slow as # of users goes up
    at first, each new user uses some otherwise idle resources
    then they start to queue
  [graph: # users, reply/sec, linear up, hits a plateau]
  [graph: # users, delay, stays at zero then linear up (queuing delay)]

How to find bottleneck?
  1. measure, perhaps find that e.g. cpu is 100% used
     but maybe not, e.g. cpu 50% use and disk 50% used, just at different
times
  2. model
     net should take 10ms, 50ms CPU, 10ms disk
  3. profile
     tells you where CPU time goes
  4. guess
     you will probably have to do this anyway
  test hypothesis by fixing slowest part of system
    can be difficult:
    if disk busy all the time, should you buy a faster disk / two disks?
      or more RAM?

You may need to make application logic more efficient
  fewer features, better algorithms
  I cannot help you with that -- application-specific
  but there are general-purpose techniques

The main performance techniques
  1. caching
  2. I/O concurrency
  3. scheduling
  4. parallel hardware (two disks, two CPUs, &c)
  these are most useful when many waiting requests
    but that will often be the case if your server is heavily loaded

I'm going to focus on disk bottlenecks
  Every year it gets harder to be CPU-bound
  what tricks for good disk performance?

hitachi 7K400: 400 GB. 7200. 8.5ms r seek, 9.2ms w.
567 to 1170 s/t. 10 heads. abt 87000 cylinders.

primer on disks
  [disk slide]
  physical arrangement
    rotating platter: 7,200 RPM, 120/sec, 8.3 ms / rotation
      continuous rotation
    circular tracks, 512-byte sectors, about 1000 sectors/track
    perhaps 100,000 tracks, concentric rings
    multiple platters, 5 for 7K400, so 10 surfaces
      cylinder: set of vertically aligned tracks
    one disk arm, head per surface, they all move together
      can only read from one head at a time
  three movements required:
    "seek" arm to desired track: varies w/ # tracks, 1 to 15 ms
    wait for disk to "rotate" desired sector under head: 0 to 8.3ms
    read bits as they rotate under head at 7200

disk performance?
  big multi-track sequential transfers:
    one track per rotation
    512*1000 / 0.0083
    62 megabytes/second
    that is fast! unlikely to be a bottleneck for most systems
  small transfers, e.g. 5000-byte web page:
    from *random* disk location
    seek + rotate + transfer
    avg seek: 9 ms
    avg rotate: 1/2 full rotation for random block
    transfer: size / 62 MB/sec
    9 + 4 + 0.1 = 13.1ms
    rate = 5000 / 0.0131 = 0.4 MB/sec
    i.e. 1% of big sequential rate. this is slow.
    sadly this is typical of real life

Lesson: lay your data out sequentially on disk!

caching disk blocks
  use some of RAM to remember recently read disk blocks
    this is often the main purpose of RAM...
  your o/s kernel does this for you
  table:
    BN DATA

```
    ... ...
  read(bn):
    if block in cache, use it
    else:
      evict some block
      read bn from disk
      put bn, block into cache
  hit cost: about 0 s to serve a small file from RAM
  miss cost: 0.010 to read a small file from disk

eviction / replacement policies
  important: don't want to evict something that's about to be used!
  least-recently-used (LRU) usually works well
    if it's been used recently, will be used again soon
  LRU bad for huge sequential data that doesn't fit
    if you read it over and over (or even only once)
    if it's been used recently, won't be used again for a while!
    don't want to evict other useful stuff from cache
    random? MRU?

how to decide if caching is likely to work?
  productive to think about working set size vs cache size
  you have 1 GB of data on disk and 0.1 GB of RAM
  will that work out well?
  maybe yes:
    if small subset used a lot (popular files)
    if users focus on just some at a time (only actively logged in users)
    if "hit" time << "miss" e.g. disk cache
    if disk I/O signif fraction of overall time
  maybe no:
    if data used only once
    if more than 0.1 GB read before re-use (i.e. people read whole GB)
    if people read random blocks, then only 10% hit probability
    if hit time not much less than miss time
      e.g. caching results of computation

i/o concurrency
  what if most requests hit but some have to go to disk?
    and you have lots of requests outstanding
  [time diagram: short short long short short]
  we don't want to hold up everything waiting for disk
  idea: process multiple requests concurrently
    some can be waiting for disk, others are quicker from cache
    can use threads
  note: threads handy here even if only one CPU
  special case: prefetch (works w/ only one thread)
  special case: write-behind

scheduling
  sometimes changing order of execution can increase efficiency
  if you have lots of small disk requests waiting
  sort by track
  results: short seeks (1ms instead of 8ms)
  the bigger the backlog, the smaller the average seek
    system gets more efficient as load goes up!
    higher throughput
```

```
  "elevator algorithm"
  maybe you can sort rotationally also, but that's hard
  cons: unfair -- increases delay for some

if you cannot improve efficiency
  buy faster CPU/net/disk if you can
  otherwise buy multiple servers (CPUs + disks) -- but how to use?
  strict partition of work: easiest
    users a-m on server 1 (cpu+disk)
    users n-z on server 2 (cpu+disk)
    no interaction, so e.g. no need for locks, no races
  hard if e.g. some operations involve multiple users -- bank xfers
    perhaps factor out computing from storage
    front end / back end
    front end can get data for all needed users from back end
    works well if CPU-bound or FEs can cache

Pragmatics
  programmer time costs a lot
  hardware is cheap
  programming most worthwhile if allows you to use more h/w

****

show slide
going to look at a quiz question from a few years ago
as practice, and to illustrate performance ideas
quizzes often present some new pretend system, ask lots of
  questions about it and effects of changes
worth practicing a bit (old exams), takes some getting used to

OutOfMoney

serving movie files, to clients, over net
single CPU, single disk
each file is 1 GB, split into 8 KB blocks, randomly scattered

Q1: 1153 seconds
seek + half rotation + (8192 / 10 MB/sec)
0.005 + 0.003 + 0.0008 = 0.0088 sec/block
times 131072 = 1153

if layout were better, how long would it take?

mark adds a one-GB whole-file cache

Q2: 1153 seconds

does that mean the caching scheme is bad?

Q3: B

what behavior would we expect to see for each of those reasons if
it were true?

in what circumstance would that caching scheme work well?
```

how could it be improved?

Threads: what is the point? do we expect threads to help?

Why do we think it is telling us where it calls yield? What is the significance of those points? (I/O concurrency)

Why might non-pre-emptive be important?

new caching code:
1. 4 GB
2. reads each block independently: *block* cache, not whole-file

Why might per-block caching be important? Probably second-order, to fix a bug in the question: that otherwise GET_FILE_FROM_DISK might prevent all other activity.

Q4: 100% hit rate.

Maybe an artifact of having sent a first non-concurrent request. What if he had started by sending many requests in parallel?

Q5: Ben is right. One CPU, threads are non-pre-emptive.

What if two CPUs, or pre-emption? Would this lock be enough?

Might need to protect all uses of cache, disk driver, network s/w, maybe best done inside each of these modules.

Might want to do something to prevent simultaneous disk read of same block, i.e. make IF, GET, and ADD indivisible. Though a spin-lock is probably not the right answer. Per-block busy flag, wait(), notify().

Q6: 0.9 * 0 + 0.1 * 1153 = 115.3

Q7: E

The cache is bigger than the total data in use, so no replacement is ever needed.

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 9: Networking Introduction

ideas for more content
  load-shedding / feedback options (pricing &c)
  why did the internet approach win?
    what was the other plan?
  voice / video on internet actually works
    despite best effort non-guarantee
    why? video is elastic.
    voice actually a lot harder despite low bitrate
  why QoS isn't needed (elastic adaptive applications?)

burstyness and flat-rate fixed-rate pipes make pricing complex
  my dsl company sells one way, buys in bulk another
  they take on a big risk -- what if i start downloading 24 hours per day?

old internet maps
  http://www.nthelp.com/maps.htm
  MCIWorldcom 2000 most hair-raising

First of 5 lectures on data networks.
  Overview today: identify the problems and design space
  Dig into details over next four lectures.
  * network are a useful systems building block
  * internal workings are a good case study of system design.
    Internet in particular an example of a very successful system.
    complex enough to be a subject of science, like the weather

What is the goal of data networking?
  Allow us to build systems out of multiple computers.
  Your problem may be too big for one computer.
  You may want multiple computers for structural reasons: client/server.
  more fundamental reason:
    A key use of computers is for communication between people, orgs.
    People are geographically distributed, along with their computers.
  So we're forced to deal with inter-computer comms.

System picture:
  Hosts. Maybe millions of them. All over the world.
  [No cloud yet, but leave room. circular hosts.]

What are the key design problems to be solved?
  I'll classify into three types:
  O: Organizational, to help human designers cope.
  E: Economic, to save operators money.
  P: Physics.

E: universality.
  Want one net, many uses
    Rather than lots of small incompatible/disconnected network worlds.
      e.g. Fedex builds its own data net for its customers to track packages
      private nets sometimes good for reliability, predictable service,
security
    The more people that use a network, the more useful it is for everybody.
    Value to me is proportional to N people using the same net.
    Value to society is proportional to N^2.

```
  What technical tools help us achieve universality?
    Standard protocols make it easy to build systems.
    But don't want to prevent evolution by freezing design w/ standards.
    Hard: standardize just what's required to make net generally useful,
      but not the things that might need to be changed later.
  One universal net means it's *not* part of each system design.
  It's a black box; simplifies design of systems that use it.
  Symbolically, a cloud that hides internal workings.
  [draw network cloud]

E: topology.
  [three new pictures; circular hosts, square switches]
  Look inside the cloud.
  Wire between every pair?
    pro: network (wires) is transparent, passes analog signals.
      It's never busy. Never any question of how to reach someone.
    con: host electronics. laying wires expensive.
  Star network. "switch". Federal Express in Memphis.
    pro: less wire cost. easy to find a path.
    con: load and cost and unreliability of hub.
  Mesh topology. Routers and links.
    pro: low wire cost. limited router fan-out. some path redundancy.
    con: shared links, higher load. complex routing.

O: routing.
  Find paths given dst "address".
  Harder in mesh than in star or hierarchy.
    [diagram: two available paths]
    change far away may require re-routing
    -> has a global element, thus scaling problems
  many goals:
    Efficiency: low delay, fast links
    Spread load
    Adapt to failures
    Minimize payments
    Cope w/ huge scale
  Routing is a key problem.

O: addressing.
  Hosts need to be able to specify destination to network.
  specifically, the address you give to the routing system
    18.7.22.69, not web.mit.edu
  ideal:
    every host has a unique ID, perhaps 128 bits assigned at random
    routing system can find address, wherever it is
    so i can connect to my PDA, whether i left it at home or in the office
  no-one knows how to build a routing system for large #s of flat addresses!
    maybe can layer on top of something else, but not directly
  In practice, encode location hints in addresses.
    hierarcical addresses, e.g. 18.7.whatever
    [diagram: 18 area, 18.7 area, ...]
    rest of inet knows only 18, not every host in 18
    Trouble if hosts move or topology changes.
    hard to exploit non-hierarchical links (rest of Inet can't use MIT-
Harvard)
  routing and addressing often very intertwined
```

E: sharing.
  Must multiplex many conversations on each physical wire.
  1. how to multiplex?
     A. isochronous.
       [input links, router1, router2, link, repeating cycle]
       reserved b/w, predictable delay, originally designed for voice
       called TDM
     B. asynchronous
       data traffic tends to be bursty - not evenly spaced
         you think, then click on a link that loads lots of big images
       wasteful to reserve b/w for a conversation
       so send and forward whenever data is ready
       [diagram: input links, router1, link, router2, output links]
       divide data into packets, each with header w/ info abt dst
  2. how to keep track of conversations?
     A. connections
        like phone system
        you tell network who you want to talk to
        figures out path in advance, then you talk
        reqired for isochronous traffic
        can allow control of load balance for async traffic
        maybe allow smaller packet headers (just small connection ID)
        connection setup complex, forwarding simpler
     B. connectionless / datagrams
        many apps don't match net-level connections, e.g. DNS lookups
        each packet self-contained, treated separately
        packet contains full src and dst addresses
        each may take a different route through network!
        shifts burden from network to end hosts (connection abstraction)

E: Overload
  more demand than capacity
  consequence of sharing and growth and bursty sources
  how does it appear?
  isochronous net:
    new calls blocked, existing calls undisturbed
    makes sense if apps have constant unchangeable b/w requirements
  async net:
    [diagram: router with many inputs]
    overload is inside the net, not apparent to sending hosts.
    net must do something with excess traffic
    depends on time scale of demand > capacity
    very long: buy faster links
    short: tell senders to slow down
      feedback
      elastic applications (file xfer, video with parameterized compression)
      can always add more users, just gets slower for everyone
      often better than blocking calls
    very short: don't drop -- buffer packets -- "queuing"
      demand fluctuates over time
      [graph: varying demand, line for fixed link capacity]
      buffers allow you to delay peaks into valleys
      but only works if valley comes along pretty quick!
      adds complexity. must drop if overload too much. source of most Inet
loss.

Behavior with overload is key.
[Graph of in rate vs out rate, knee, collapse.]
Collapse: resources consumed coping with overload.
This is an important and hard topic: congestion control

E: high utilization.
How much expensive capacity do we have to pay for?
Capacity >= peak demand would keep users happy.
What would that mean?
single-user traffic is bursty!
[typical user time vs load graph -- avg low, peak high]
Worst case is 100x average, so network would be 99% wasted!
e.g. my one-megabit DSL line is almost entirely idle
too expensive in core to buy Nusers * peak
But when you aggregate async users -- peaks and valleys cancel.
[a few graphs, w/ more and more users, smoother and smoother]
peak gets closer and closer to average
100 users in << 100x capacity needed for 1 user.
less and less idle capacity, lower cost per bit transmitted
"Statistical multiplexing."
Hugely important to economics of Internet
Assumes independent users
mostly true -- but day/night, flash crowds
50% average daytime utilization maybe typical on core links

Once you buy capacity, you want to get as close to 100% as possible.
fixed cost, want to maximize revenue/work
may need feedback to tell senders to speed up!
want to stay at first knee of in/out graph.
Or maybe less to limit delay
[graph: load vs queuing delay]

P: errors and failures.
Communication wires are analog, suffer from noise.
Backhoes, unreliable control computers.
General fix: detect, send redundant info (rxmt), or re-route.
How to divide responsibility?
You pay your telecommunications provider, so they better be perfect?
Leads to complex, expensive networks: each piece 100% reliable.
You don't trust the network, so you detect and fix problems yourself.
Leads to complex host software.
Decision of smart network vs smart hosts turns out to be very important.
Locus of complexity pays costs but also has power.
"Best design" depends on whether you are operator or user.

P: speed of light.
foot per nanosecond.
14 ms coast to coast, maybe 40 ms w/ electronics.
Request/response delay. Byte per rtt -> 36 bytes/second.
[picture of pkt going over wire -- and idle wire]
Need to be more efficient.
Worse: delay hurts control algorithms. Change during delay.
Example: Slow down / speed up.
Result: Oscillation between overload and wastefully idle.

O: wide range of adaptability.

One design, many situations.
Applications: file xfer, games, voice, video.
Delay: machine room vs satellite.
Link bit rate: modem vs fiber optics.
Management: a few users, whole Internet.
Solution: Adaptive mechanisms.

Internet in design space
Asynchronous (no reservations)
Packets (no connections)
No b/w or delay guarantees
May drop, duplicate, re-order, or corrupt packets -- and often does
Not of much direct use! End hosts must fix
but gives host flexibility, room for innovation
"best effort"

Next lecture: start to talk about solutions.

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 10: Layering, Link Layer

Let's design a data network.
  How do we organize our design?

Reminder: mesh networks.
  [draw host/switch/mesh picture]

Idea: protocols.
  Two entities (peers) are talking.
  Protocol formally defines structure of conversation.
  Typically sequence of messages -- packets.
  Format:
    Specific set of message types.
    What does each bit of the message mean?
    [example: ethernet paper, dst, src, data, checksum]
  Rules for what happens next, state machines.
  Semantics: what does it mean?
  Often you can learn all you need from the formats...

But there are many levels of communication.
  [draw the on *mesh* net, nested...]
  Analog waveforms on a wire.
  Messages from one switch to the next.
  Similarly for host / switch.
  Application to application.
  There are many interacting protocols here.
  How do we organize them?

Idea: layers.
  Intuition: protocols nest.
  Inner protocols building blocks for outer ones.
  Let's formalize that way of organizing multiple protos.
  Choose and define a useful protocol. [box <--> box]
  Define s/w interface so other layers can use it. [stack up]
  It may in turn use more primitive layers. [stack down]
  Can build up functionality this way.
  But use modules, abstraction to control complexity.
  Hard part: choosing useful layer boundaries.

6.033 layer model:
  [draw stacks for host, switch, host]
  Physical: analog waveforms -> bits.
  Link: bits -> packets, single wire.
  Network: packet on wire -> packet to destination.
  End-to-end: packets -> connections or streams.
  Application.
  physical almost always tightly bound to Link.
  And application isn't a generally useful tool.
  Note: layer may have many clients
    multiple apps using e2e, multiple e2e using net
    need a way to multiplex them
  Note: net layer may use multiple links
  So the real picture in one host
    app1 app2 app3
      TCP UDP

```
        IP
     Eth WiFi
  Layers == outline of data networking topic.

Stack of layers:
  Repeated scheme for layer interaction
  Each layer adds/strips off its own header.
    Encapsulates higher layer's data as "payload".
    [add to pkt diagram; interior header &c]
  Each layer may split up higher layer's data.
    [stream split into payloads of packets]
  Each layer multiplexes multiple higher layers.
    [put protocol # field into packet]
  Each layer is (mostly) transparent to higher layers.
    data delivered up on far side == data in

Physical Layer
  Built out of wires.
  We will only talk about one direction.
  Just build two for bi-directional communication.
  [picture: Sender, wire, Receiver]
  can send "signals": voltages, optical power levels.
  What we want is a stream of bits.
  problems:
    which value, 0 or 1? vs noise
    when to sample at receiver

Straw man one:
  3 wires: data bit, ready, ack
  [picture]
  speed of light limits data speed.
    coast-to-coast: 30 bits/sec
    one mile: 200 kbits/sec

Straw man two:
  Assume sender and receiver have accurate oscillators.
    I.e. same frequency.
  [draw the two oscillators]
  Sender sends one bit per clock period.
  [picture of waveform. mark sampling spots.]
  Easy problem: phase changed by wire delay.
  Hard problem:
    You can probably build one accurate clock.
    You probably can't build two that agree.
    You can never assume synchronized clocks in dist systems.

Straw man three:
  Send the clock on a separate wire?
  Expensive, skew between wires if fast or long

The right answer:
  Idea: clock recovery from the signals themselves.
  Receiver keeps a local clock.
  Adjusts it if signal transitions aren't on clock transitions.
  balanced signals: you need transitions! no strings of zeroes.
  So can't directly encode 0/1 as high/low signals.
```

```
  Ethernet/Manchester has four wave-forms == "symbols"
    LH: 0 bit
    HL: 1 bit
    LL: idle
    HH: unused
    so only 2 of 4 codes used. [picture]
  Could be more efficient: 8 symbol values per 2 bits. [picture]
    LLH: 00
    LHL: 01
    HLL: 10
    HLH: 11
    others unused, e.g. LLL, to aid clock recovery
  Why not 128 signals per 127 bits?
    single signal error wrecks 127 bits...

Could we declare this to be a layer?
  Is it a generally useful abstraction?
  No: no way to share. Just bits.

Link Layer: Bits to Packets/Frames
  Why packets? Why not continuous data?
    Need to inter-mix packets from different conversations.
  How do we delimit packets?

Packet framing
  Link has two states: idle, sending data.
  Need to signal the state transitions.
  Idle -> Sending (start of packet).
  Sending -> Idle (end of packet).
  Let's signal idle with all low signals.
    LL not used by Manchester to send bits
  Start of packet easy: just send a one bit.
    Might want some longer alternating sequence for clock recovery.
    e.g. 8x LH followed by HL, in case rcvr misses first few LH
    Called a preamble.
    [packet picture: preamble, data, end-mark]
  mark end of packet?
    special code that cannot ordinarily appear in the data stream.
    transform data stream to eliminate magic bit sequence.
    OR two zero signals in manchester scheme.
  what if end-of-packet signal corrupted? wreck next packet?
  why don't we use a length field at start instead of a special code?

Can we put an interface here, and make a layer?
  Not yet: how do we know whose packet is whose?
  Really indicating which layer above us.
  All we need is a number in the packet header.

We could turn this into a layer.
  By convention, link layer detects errors.

What should we do about errors?
  E.g. noise corrupts some bits.
  We need to define the link layer abstraction.
  What can it reasonably *guarantee*?
  Could just ignore the problem, let higher layer deal.
```

    Careful higher layers probably check anyway.
  Can we guarantee to deliver all packets perfectly?
    Hard for link layer to make this guarantee.
    Ask previous switch to re-send? Complex, buffering.
    What if switch failes? => link cannot guarantee!
    Maybe app doesn't care anyway, e.g. real-time voice.
  Realistic answer:
    Guarantee that all packets delivered are correct.
    We're allowed (expected) to discard corrupted packets.
    Example of "best effort" contract.
    This means link layer just *detects* bit errors.

Error detection
  best plan depends on error patterns.
    single bit due to short noise.
    microwave oven wipes out many packets.
    coding wipes out one block of e.g. 2 bits.
  example: xor byte at the end.
    [picture: vertical column of bytes, xor at bottom]
    detects any one bit error, some multi-bit errors.
    doesn't detect e.g. byte exchange.
    much more sophisticated schemes available!
    usually called "checksums".

Error correction
  A single bit error causes link layer to discard whole packet.
  Seems wasteful.
  We can often squeeze out a little more performance.
  Here's a plan for correcting one-bit errors.
  [add a 9th bit to each byte.]
  Still have to discard if there are two incorrect bits.

What does our link-layer protocol look like?
  [refer back to Protocol board]
  Format: mostly framing, also proto #.
  What's the abstraction?
    exposes packets. (not e.g. files...)
    allows sharing among higher protocols.
    best-effort: may not deliver a packet.
    will rarely give you a corrupted packet.
  Ethernet MAC operates at about this level.

Now we can move a packet along a single wire.
  Next lecture: move packet along a path of switches.

6.033 Computer System Engineering
Spring 2009

Lecture 11

Sam Madden

Network Layer

Today: Network layer.

Job of network layer is to find and forward packets along a path between sender and receiver in a multi-hop network.

Show example network:

"routers"

Network layer interface

Two main jobs:

Forwarding -- sending data over links according to a *routing table*
Routing -- process whereby routing tables are built

Forwarding:

Show routing table in network

Point out that many routing tables are possible (example)

Show stack -- annotate with "net_send" and "link_send" and "net_handle" and "e2e_handle" calls, encapsulation

Show link selection in stack

Forwarding -- mechanical. Just perform a lookup in a table.

Pseudocode.

```
forwarding_table t

net_send(payload, dest, e2eprot):
        pkt = new packet(payload,dest,e2eprot)
        net_handle(pkt)

net_handle(pkt):
        if (pkt.dest == LOCAL_ADDR):
                e2e_handle(pkt.payload, pkt.e2eprot)
        else:
                link_send(t[p.dest].link, pkt)
```

Routing -- compute the forwarding table

How to compute forwarding table? Manually -- not scalable.

Centrally -- not a good idea (why?)
- need a routing algorithm to collect
- collection requires many messages
- hard to adapt to changes

Path Vector Algorithm -- Distributed

Each node maintains a forwarding table T , with:
"e2e_handle" calls, encapsulation

Dest	Link	Path

Two steps:
- advertise (periodic)
  - send T to neighbors

integrate(N,neighbor, link) -- on receipt of advertisement from neighbor
merge neighbor table N heard from neighbor on link into T

Merging:
for each dest d w/ path r in N:
if d not in T, add (d, link, neighbor ++ r) to T
if d is in T, replace if (neighbor ++ r) is shorter than old path

Example:

(If everybody picks best path to every dest, you can see that for a network with most distant nodes separated by N hops, in N rounds everyone will know how to reach everyone else in N steps.)

Q: what is the purpose of keeping the path in the table?

Problems:

- permanent loops?
  - won't arise if we add a rule that we don't pick paths with ourselves in them; this is what we need the path for!

- temporary loops -- arise because two nodes may be slightly out of date
  example
  - soln: add send count -- "TTL" -- to packet

- failures / changes -- repeat advertisements periodically,
  remove paths in your table that aren't re-advertised
  (e.g., a path P that begins with router R should be in the

next advertisement from R.)

- graph changes -- same as failures

How does this work on the Internet:

At first, internet was a small network like this

Show evolution slides

What is the problem with using path vector here?

Network is huge

> 1 B nodes on network

Even if we assume most of those are compters that connect to only their local router (so don't really need to run the path vector protocol), there are still many millions of routers in the Internet

Each router needs to know how to reach of these billions of computers

With pure path vector, each node has a multi-billion entry table (requiring gigabytes of storage)

Each router has to send these gigabyte tables to each of its neighbors; millions of advertisements propagating around. Disaster.

Solution: hierarchical routing

Subdivide net into areas; with multiple levels of routing

One node representative of each area; perform path vector at area level. Within each area, free to do whatever. (For example, use more hierarchy.)

(e.g., a path P that begins with router R should be in the

How to name nodes:

area.name

On Internet, this is IP address

E.g., 18.7.22.69 -- this is mit.edu

Internet routers running -- BGP -- advertise prefixes of these address

Show advertisements (e.g., "18.*.*.*"...)
17.1*.*.*

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 12

Sam Madden

End to End Layer

Last time: network layer -- how to deliver a packet across a network of multiple links

Recall that network layer is **best effort**, meaning:

- packets may be lost, reordered, garbled
- losses due to no route, queue overflow, packet corruption / collisions, etc.
- reordering due to packets taking different routes, retransmissions

Network layer is also message oriented.

E2E layer addresses these limitations. Some things applications might want:

1. Connection, stream of data, rather than having to explicitly divide data into messages

2. All packets to arrive (reliable)

3. No duplicates (at most once)

4. Packets arrive in order

5. Efficiency

Not all applications need all of these -- for example, in voice chat app, you might prefer more predictable latency w/ some packets being dropped, rather than all packets arriving with unpredictable latency.

This list above is roughly what Internet's TCP gives you. Internet provides other protocols, like UDP (none of the above) or RTP (streaming, but not reliable)

Different E2E layers will provide different subsets of these features. Goal is to build these features without modifying network layer.

E2E Interface

Diagram: (including port->app table), packets

Streaming connection (add entries to port->app table)

src	dest
conn = open_stream(dest, port)	conn = listen(port)
send(conn, bytes1)	
send(conn, bytes2)	bytes = recv(conn)
close_stream(con)	

E2E layer fragments streams up into packets that it sends to network layer.

Reliability --> At least once

All packets arrive, and no packets are corrupted

For corruption, typically packets have some kind of checksum, which is checked when the packet is received.

How to ensure all packets arrive?

Idea:

have receive send an acknowledgement for every packet. Set a timer on the sender, and when the ack doesn't arrive after that much time, resend.

associate a unique sequence number with each packet.

attach sequence number to the ack.

Diagram:

How to set the timer interval? 1 s? 1ms? Networks vary a lot! In a local wired network, might see e2e latencies < 1ms. In a cellular network, or intercontinenal network, might see latencies > 1s.

Too short --> always resending
Too long --> underutilization of network (show)

So what should we do? Adapt timeout!

Measure round trip time (RTT) and adjust.

RTT = time between packet sent and ack received
Just use one sample? No.
Just use last time + error? No. -- too much variability, even on one network (show slides)

Use some average of recent packets.

To avoid having to keep window of averages around, can use exponentially weighted moving average:

(show slides)

Ok, so now we set the timer appropriately. Ensures that at least one copy of every data item arrives.

Can we have duplicates? (yes, why?)

What to do about this?

At most once delivery

On receiver, keep track of which packets have been ack'd (in a list). When a duplicate packet arrives, resend ack, but don't deliver to app.

Diagram:

When can you remove something from list of acks? Since acks can be delayed or lost, could receive resent packets quite a bit later. One typical solution is to attach last msg

id received on messages from sender.

At least once delivery and at most once delivery together are **exactly once delivery**.

Bit of a misnomer.

Issues:

- Suppose receiver crashes after receiving message but before sending ack; reboots --> did it process the message or not, what if it receives message again?

We will talk much more about building reliable systems and these issues after spring break.

- What does an ack mean? Typically just that the E2E layer handed the packet off to the application, *not that the application processed the message.*

(Skip?) How big to make sequence numbers?

16 bits? In gigabit net, with 1 kbit packets, will send 1 million packets / sec. So will exhaust sequence number space in about 60 ms. Packets could easily be delayed this long.

32 bits? 4000 seconds. Better, but some connections will last this long. Need to recycle sequence numbers.

Note that the protocol thus far deliver stuff in order b/c it only has one outstanding message at a time.

At this point, we've built a streaming, exactly once, in order delivery mechanism. Problem is it is SLOOW.

Show diagram:

Suppose RTT is 10 ms. Can only send 100 packets / second. If packets are 1000 bits, then we are only sending at 100 Kbit/sec. Not good!

Can we just make packets bigger?

How big would they have to be for gig-e? 10,000 times bigger --> 10 mbits. Huge! Retransmissions are super expensive, and we aren't doing a very good job of multiplexing the network.

Solution: smaller packets, but multiple outstanding at one time.

"window" of outstanding packets

Show diagram:

Still wasteful, since sender waits an RTT. Solution: slide window:

Note that if the window size is too small, may still end up waiting.

So, how big to make windows? Want to be able to "cover" delay.
Suppose we have a measure of RTT.

And suppose we know the maximum rate the network can send (either because of links, delays in the network, or limits at the sender or receiver),

Max packets outstanding is then

window = RTT x rate

"bandwidth delay product"

Of course, this assumes we can measure the rate accurately, which turns out to be hard because rate depends on what is happening inside the network (e.g., what other nodes everywhere else inside the network is doing.) This is the topic for next time.

Acks and windows -- two possibilities:

- acks are per-message, as above.
    + precise (only retransmit one packet)
    - can resend data that doesn't need to be resent if ack is lost

- acks are cumulative
    e.g., indicate the sequence number up to which all packets have been seen
    + one ack can cover many packets (fewer acks)
    - sender doesn't know exactly what was lost, so has to retransmit a lot more

In practice, use both:
    - cumulative in common case (e..g, every n packets)
    - selective when there are single losses

Out or order delivery

Even though all packets arrive, packets may not arrive in order, because of windowing

Solution: re-ordering buffer.

Fixed size buffer at receiver recording out of order packets. If a packet arrives in order, deliver to app. Otherwise, add to out of order buffer while we wait for more recent packet to arrive. Diagram:

When a missing packet arrives, go ahead and send prefix of buffer to app.

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 13

Sam Madden

Congestion Control

Last time, we saw how the end to end layer ensures exactly once delivery, using windows to efficiently make use of the network.

Couple of loose ends:

- Reordering -- packets may arrive at the receiver out of order

```
recv(p)
        slot = p.sno - last
        if (slot > last + size)
                drop
        else
                new = slot is empty
                if (new) put p in slot
                ack p
        if (slot == last and new)
                deliver prefix to app
                slot += size(prefix)
```

Cumulative acks

So far, have presented acks as being *selective* -- for a specific packet

In Internet, sometimes *cumulative acks* are used

in figure above, when p1 arrives, would send ack(2), indicating that all packets up to 2 have been received.

+ insensitive to lost acks
    (if using selective acks, a dropped ack requires a packet retransmit. Here, if the ack for P2 was lost, when P0 arrives, receiver knows P2 arrived.)

- sender doesn't know what was lost

    here, sender has sent p3, p4, and p5; p4 and p5 have been receiver, but only receives an ack for up to p2.

    can lead to excessive retransmission

In the TCP, both cumulative and selective acks are used:

    common case, use cumulative, so that a lost ack isn't problematic
    when acking a packet for which later packets have arrived,
        use selective acks, so that sender knows what has arrived

Choosing the window size

Last time, we send that the window size should be:

    Window = Rate x RTT

Where Rate is the maximum rate the receiver can accept packets at

This will keep the receiver busy all the time

E.g., suppose receiver can handle 1 packet / ms , and RTT is
    10 ms; then window size needs to be 10 packets for receiver to always be busy

2 questions:

1) How does the receiver know how fast it can process packets?
2) What if the network can't deliver packets as fast as the receiver can process?

1) It doesn't. In practice it doesn't matter that much -- just set it to be big enough. For example, on 100 Mbit ethernet, with 10 ms RTT,

window = 100 Mbit/sec * .01 sec = 1 Mbit = 128 KB

So what if the network can't handle packets at this rate -- is that a problem?

Suppose sender and receiver can both communicate at 3 pkts / sec, but that there is a bottleneck router somewhere between the sender and receiver that can only send 1 pkts / sec. Suppose RTT is 2 sec (so receive window is 6 pkts)

After 1 sec, there will be 2 packets in this router's queues
After 2 seconds, there will be 4 packets

Congestion collapse = delivered load constantly exceeds available bandwidth

Notice that if we could adapt RTT on sender fast enough, that would help, since we'd sender fewer duplicates.

But we could still get into trouble if we have a big window and there is a slow bottleneck.

Why does congestion arise?

Routers in the middle of the Internet are typically big and fast -- much faster than the end hosts. So the scenario above shouldn't arise, should it?

Issue: sharing. Sources aren't aware of each other, and so may all attempt to use some link at the same time (e.g., flash crowds). This can overwhelm even the beefiest routers.

Example: 9/11 -- everyone goes to CNN. Apple -- releases new products.

What do do about it:

Avoid congestion:

Increase resources? Expensive and slow to react; not clear it will help

Admission control? Used in telephone network. Some web servers do this, to reduce load. But its hard to do this inside of the network, since routers don't really have a good model of applications or the load they may be presenting.

Congestion control -- ask the senders to slow down.

How do senders learn about congestion?

Options:

1) Router sends feedback (either directly to the sender, or by flagging packets, which receiver then notices and propagates in its acks.)

Works, but can be problematic if network is congested because these messages may not get through (or may be very delayed)

2) Assume that packet timeouts indicate congestion.

What to do about congestion:

- Increase timeouts.
- Decrease window size.

Need to do this very aggressively -- otherwise it is easy for congestion to collapse to arise. CC is hard to recover from.

TCP does both of these:

Timeouts: exponential backoff

On retransmitted packet, set:

set timeout = timeout * c ; c = 2

up to some maximum

timeout increases exponentially

After ack's start arriving (losses stop)
, timeout is reset based on RTT EWMA (as in last class)

2 windows: receive window + "Congestion Window":

Window size = min(congestion window, receiver windoow)

On retransmitted packet:

CW = CW/2

(Min size = 1)

On ack'd packet:

CW = CW + 1 (up to receive window size)

To initialize window quickly, use TCP "slow start":

At the start of the connection, double CW up to receive window, unless losses occur:

Slow start -- while connection is starting only

On each ack:

CW = CW * 2 (max receive window)

Is TCP "fair"?
no ; applications aren't required to use TCP in which case they won't obey congestion rules. Applications can also use multiple TCP connections.

So why does this work? Apparently because most traffic is TCP.

6.033 Computer System Engineering
Spring 2009

DNS, CDNs Lecture 14

Sam Madden

DNS

What is the relationship between a domain name (e.g., youtube.com) and an IP address?

DNS is the system that determines this mapping.

Basic idea:

You contact a DNS server, give it a query

It responds with the answer, or forwards your answer on to some other server

Example:

Whenever a server answers a query, it adds it to a cache

When you get a response, you add it to a cache

Each cached value has a duration after which you are supposed to look up the name again

At the top of the DNS hierarchy are root servers

Then servers for .com, .edu, etc ...

Then servers for MIT, youtube, etc....

Show cached

dig +norec csail.mit.edu

A record vs NS Record
TTL

Observe that A records expire much sooner than NS Records

Example:

csail.mit.edu

dig
dig edu@E.ROOT-SERVERS.NET.
dig mit.edu @G.GTLD-SERVERS.NET.
dig csail.mit.edu @bitsy.mit.edu.

How do you get your DNS server initially?

- Your admin tells it to you
- DHCP (Domain Host Configuration Protocol) -- you send out a broadcast on the local link -- DHCP server responds (via another broadcast message) with:

    - An IP address for you to use
    - The IP address of a "gateway" router for you to use
    - The IP address of a DNS server to use
    - ....

DNS has some fancy features:

- One physical machine can have multiple names

- One name can correspond to multiple IP addresses
    DNS load balancing
    DNS server picks which name to return
        most DNS servers cycle through these in "round robin" fashion

Example:

dig google.com

;; ANSWER SECTION:
google.com. 282 IN A 209.85.171.100

google.com. 282 IN A 74.125.45.100
google.com. 282 IN A 74.125.67.100

Content delivery networks (CDNs -- e.g., Akamai)

Use DNS to provide scalability and adapt to load

Suppose I have some content that is accessed a lot -- e.g., a video on youtube

I can balance load amongst my local servers using DNS load balancing, but I still have to own the servers.

Puts a huge load on my servers to deliver it; can't adapt to load spikes (e..g, the "slashdot" effect.)

Also, for users that are far away (e.g., in Asia or Australia), they have to download that content over long distance and thin pipes, and ISPs in Australia have to pay a lot for than bandwidth.

For content -- like this video -- that is accessed repeatedly, would be be better to not have to go all the way to San Bruno CA everytime.

Idea: create a local cache

Solution 1: Proxy cache. For every URL, look up in a local cache (perhaps run by your ISP) to see if the content is there. If it is, fetch it. Otherwise, get original data. Just like DNS requests, web pages can have cache lifetimes associated with them which proxy caches respect.

Problems:

- Helps ISP but not necessarily content provider
- requires clients to configure their browsers to use proxy caches
- Doesn't address slashdot effect

Exist so called "transparent proxies" that can do this filtering automatically, but this may be distasteful to users, especially if their ISP is doing it (companies do this all the time.)

Solution 2: Content Distribution Networks (e.g., Akamai)

Show diagram:

Give example:

nytimes.com

dig graphics8.nytimes.com

From both MIT network and cellular network

Observe that TTL for answer is very short -- Why? -- Handle slashdot effect -- can dynamically start using more and more akamai servers for a particular request

Akamai -- company

Gives a way for content provider to offload load from server

Also helps ISP if server is inside ISP (creates an incentive for ISPs to participate!)

Akamai has thousands of caches all over the world

When a request for content -- like from images8.nytimes.com arrives -- it uses dns to forward user to the nearest server.

Determining the "nearest" server and how many servers to allocate is their secret sauce.

For dynamic content, Akamai also works.

Example -- show attempt to select best route back to Akamai, maintenance of reachability info, etc.

Akamai is a more general example of something called an **overlay network**.

An overlay is a way to create a network with new features or a different structure by building on top of an existing network.

Akamai creates an overlay on top of the IP network that chooses the best route from amongst a collection of IP servers.

Overlays are used widely to extend the network with new features, for example :

Provide a different topology (e.g., direct connection of clients)
- E.g., for administrative reasons (VPNs)
- Or for performance reasons (Akamai)

Provide a different addressing mechanism (e.g., content addressability, P2P)

VPN example:

Companies want to allow remote users or sites to have access to their corporate Internet sites

Good old days might have done this with a modem, but that's slow, and a pain to run.

VPN provides a way for remote users to appear to be on internal network while actually being external.

Idea is to "tunnel" traffic over public internet using an overlay (client and server inside of network):

Diagram:

This is a simple example, but in principle can create complex multi-hop VPNs.

6.033 Computer System Engineering
Spring 2009

Lecture 15 Building Fault Tolerant Systems from Unreliable Components

Sam Madden

So far what have we seen:

Modularity
- RPC
- Processes
- Client / server

Networking
- Implements client/server

Seen a few examples of dealing with faults -- e.g., providing exactly once semantics in a best-effort system.

Goal of this part of the course is to understand how to systematically build reliable systems from a collection of unreliable parts.

Basic way this is going to work is to introduce redundancy, e.g.:

- in TCP this takes the form of sending data multiple times;
- today: a general technique -- replication of critical system coponents
- next time: in a file system or database it often takes the form of writing data in more than one place.

Some high level points:

- These techniques improve reliability but building a perfectly reliable system is impossible. Best we can do is to get arbitrarily close to reliable. Our guarantees are going to be essentially probabilisitc.

- Reliability comes at a cost -- hardware, computational resource, disk bandwidth, etc. More reliability usually costs more. Have to decide -- as with TCP -- whether the additional reliability is worth the cost. Always a tradeoff.

This lecture:

Understand how to quantify reliability

Understand the basic strategy we use for tolerating faults

Look at replication as a technique for improving reliability / masking faults

Going to do this in the context of disk systems, but many of these lessons apply

to other components. (Will come back to this at the end.)

Why disks?

Diagram (disk, cable, computer, controller, file system sw)

Where do failures happen?
(Everywhere.) Show slide.

Fault anywhere in this system makes compute unusable to user.

Goal is to improve the general availability of the system.

Measuring Availability

Mean time to failure (MTTF)

Mean time to repair (MTTR)

Availability = MTTF / (MTTF + MTTR)

MTBF = MTTF + MTTR

Suppose my OS crashes once every month, and takes 10 minutes to recover
MTTF = 720 hours (43,200 minutes)
MTTR = 10
43200 / 43210 = .9997 ("3 nines") -- 2 hours downtime per year

General fault tolerant design process:

- Quantify probability of failure of each component
- Quantify costs of failure
- Quantify costs of implementing fault tolerance

Probabilities here are high for many components, but

HOWEVER, Unlike other components, disk is the one that if it fails your data is gone -- can always replace the CPU or the cable, or restart the file system.

You can replace the disk, but then your system doesn't work. So cost of disk failure is high.

Ok, so why do disks fail?

show disk slide

spinning platters
magnetic surface
data arranges in tracks and sectors
heads "float" on top of surface, move in and out to r/w data

what can go wrong?

whole disk can stop working (electrical fault, broken wire, etc.)

sector failures:

can read or write from the wrong place (e.g., if drive is bumped)
can fail to read or write because head is too high, or coating too thin
disk head can contact the drive surface and scratch it

previous slide suggests disks fail often. what does the manufacturer say?

show Seagate slide

Mean time between failure : 700,000 hours
80 years? 80 years ago we were programming with abacuses.

What's going on here? How did they measure this?

Ran, say, 1000 disks for say 3000 hours (3 million hours total)
Had 4 failures
So claimed that they 1 failure every 750,000 hours

What does the actual failure rate look like? Bathtub (5 years)

So what does this 700,000 hours actually tell us? Probability of failure in the bottom of the tub. Helpful if I am running a data center with a thousand machines and want to know how frequently I will have to replace a disk. About 1 % failure rate per disk per year.

Would buying a more expensive disk help?

not really (show slide) -- double the MTTR, 5x the price

Coping with failures:

1st type of failure -- sector errors

What can I do

(show arch diagram -- with disk, firmware, driver, FS)

1) Silently return the wrong answer

2) Detect failure --

each sector has a header with a checksum
every disk read fetches a sector and its header
firmware compares checksum to data
returns error to driver if checksum doesn't match

3) Correct / mask

Re-read if firmware signals error (if an alignment problem)
(Many other strategies)

These are general approaches -- e.g., could have checksums at the application level (in files), or a backup to recover / correct from a bad file.

Disks have checksums -- how much do they help?

(Show SIGMETRICS slide)

1.7 million drives
10% of drives develop errors over 2 years

These are unrecoverable read failures
Would have resulted in bad data going to file system

Do disks catch all errors? (Show FAST08)

1.5 million drives, 41 months, 400,000 corrupt reads on 1000 disks
(Don't know how many reads were done, but each drive probably read millions of blocks -- so we are talking trillions of reads.)

Very small fraction of uncaught failures.

But they do happen; however, many systems assume they don't (e.g., your desktop file system.)

What about whole disk failures?

Powerful solution: replication

Write everything to two or more disks

If read from one fails, read from the other

Diagram:

```
write (sector, data):
        write(disk1, sector, data)
        write(disk2, sector, data)

read (sector, data)
        data = read (disk1, sector)
        if error
                data = read (disk2, sector)
                if error
                        return error
        return data
```

Does this solve all problems? (No)

Uncaught errors

Can do voting / TMR -- diagram

Other parts of the system that fail

Show Non-stop Slide

Replicate everything, including operating system -- run N copies of every application, compare every memory read / write

Cost is huge, performance is pport

Assumes failure independence

(what about a power failure)

Not what app wants -- application is writing large data items that span multiple blocks; what if there is a power failure halfway through those write

Summary

Disks:

lifetime about 5 years

1 %/ year random total failure

10% of disks will develop unreadable sectors w/in 2 years

small fraction of uncaught errors

Replication is a general strategy that can be used to cope

URLs:

http://h20223.www2.hp.com/NonStopComputing/downloads/KernelOpSystem_datasheet.pdf

HP Integrity NonStop NS16000 Server data sheet - PDF (May 2005)

http://www.usenix.org/events/fast08/tech/full_papers/bairavasundaram/bairavasundaram.pdf

http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_7200_10.pdf

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 16 -- Atomicity

Sam Madden

Last week: saw how to build fault tolerant systems out of unreliable parts

Saw how a disk can use checksums to detect a bad block, and how we can use replication of disks to mask bad block errors.

Goal today is to start talking about how we can make a a multi-step action look like a single, ***atomic action***.

Why is this important? Example>

Two bank accounts, balances stored on disk

Transfer routine

```
xfer(A, B, x):
        A = A - x  //writes to disk
                        <---------------- crash!
        B = B + x // writes to disk
```

Lost $x. Note that replication doesn't fix this (unless I can prevent the system from crashing completely.) Even if the write to A happens just fine, we've still lost x dollars.

Notice that what has happened is we've exposed some internal state of this action that should have been atomic.

What we'd really like is some way to make this action either happen or not happen. E.g., to preserve money.

This is called **"all or nothing atomicity**" -- either it happens or it doesn't.

Related problem: **concurrency**. Suppose two people run:

```
A = 10;  B  = 20
U1                      U2
----                    ---
transfer(A,B,5)         transfer(A, B, 5)
```

suppose A = A-X is actually

```
        R0 = read(A)
        R0 = R0 - X
```

write(A, R0)

Outcomes:

A = 0, B = 30 (correct)

U1	U2
RA	
	RA
	WA
WA	

A = 5 (not correct!)

These two actions weren't isolated -- they saw each other's intermediate state.

Should have only allowed A = 0, B = 30.

Isolation goal: outcome of concurrent actions is equivalent to some serial execution of those actions.

E.g., A & B == A then B or B then A.

Note that we have already seen a way to do this with locks, but that requires complex manual reasoning. For example, suppose a user issues a series of transfer that should be done atomically:

```
transfer(A,B)
transfer(C,D)
transfer(D,A)
```

can't just push locking logic into transfer -- procedure that does transfer has to do locking to provide isolation.

Goal is to develop an abstraction that provides both all-or-nothing atomicity and serial equivalence for concurrent actions.

This abstraction is called "atomic actions" or "**transactions**"

Works as follows:

T1	T2
begin	begin
transfer(A,B,20)	transfer(B,C,5)
debit(B,10)	deposit(A,5)
...	....
end	end
aka "**commit**"	

Properties:

- User doesn't have to put explicit locking anywhere.

- All or nothing

- Serial equivalence

- No need to pre-declare the operations -- things become visible when "end" is issued.

Extremely powerful abstraction. Tricky to implement, but makes programmers life easy.

Most common system that uses these techniques is a database. Rather than representing data as abstract files, it represents data as tables with records.

For example:

Accounts:

Customers:

...

Can write transactions that read and modify a mix of multiple tables, e.g.

- complex transfer ops between accts or
- delete a customer and all of his accounts.

...

Today -- start talking about implementing all-or-nothing property without isolation (which we will address in a couple of lectures.)

Suppose we've got a file of database balances (e.g., an excel spreadsheet) stored in a single file

Imagine that the file is loaded into memory, and that the "save" operation works as follows:

```
save(file):
        for each record r in file:
                if r is changed
                        write b to file
```

Q: what is the problem with this?

might get have way through writing file and crash, then file is in some intermediate state

version 2 ("shadow copy")

```
save(file):
        copy(file,fnew)
        for each record r in fnew:
                if r is dirty, write r o fnew
        move(fnew, file) //rename  --- commit point!
```

requires an atomic rename operation

after crash, either have old version or new version

atomic rename:

```
        unlink(file)
                                <--- crash!  (file disappears)
        link(fnew,file)
        unlink(fnew)
```

actual:

```
        link(fnew, file)
                                <-- crash!  (two copies of file!)
        unlink(fnew)
```

link command is the "commit point" -- after it happens, atomic action is guaranteed to finish in it's entirety (the "all" part of "all or nothing")

File system state on disk

directory block:

name	inode#
file	1
fnew	2

inode 1:
    blocks: A, B, C
    refcount: 1

inode 2:
    blocks D, E, F
    refcount: 1

link:

                                    <---- crash!   //both file and fnew exist, but different
(CP) modify directory block   //both fnew and file point to new file
                                    <---- crash!   //refcount wrong
    update refcount

unlink:

                                    <---- crash! //both fnew and file exist, refcount wrong
    modify directory block
                                    <---- crash! // only file exists, refcount wrong
    update refcount

assume a single disk write is atomic -- disks mostly implement this (e.g., using a capacitor that "finishes" a write)

will see the details of how files systems can ensure this property without this assumption in Thursday's recitation (or see the notes -- section 9.2 discusses); basic idea is to write two copies of the block, return the one that is valid using checksums

"recover": (restart processing to ensure all or nothingness)
    scan fs, fix refcounts
    if exist(fnew) and exist(file)
        remove(fnew) // file is either old or new

Q: when is the commit point? just after the modify directory block in link

some way towards our vision of atomicity; problems:

- have to copy the whole file every time, even if we only modified a few records

- only works for one file -- what if changes span multiple files?
  (e.g., both customer and accounts table)

  - could arrange all the databases you want to edit in a directory, atomically rename new directory

  - requires fixes directory structure beforehand

- unclear how concurrency works. can achieve isolation if we only allow one user to access at a time, but that's very restrictive.

(we would rather allow different uses to concurrently access different parts of a file)

Going to address these limitations next time, but before we do some key features of this approach:

- we write two copies of the data
- switch to new copy
- remove old copy

All of our atomicity schemes are going to work like this --
Golden rule of recoverability:

NEVER OVERWRITE ONLY COPY OF DATA.

Next time we will develop a general atomicity preserving scheme that works for multiple objects.

Doesn't require us to copy all blocks of an object -- only those we modify.

Idea is to write a log of each change we make, reflecting the state of the object that was changed before and after the change. Can then use this log to go from any intermediate state to the "before" state or the "after" state.

E.g.,: Database A, Database B; A10 v0 B7 v0
Before:

A10 v0 B7 v0

After:

A10 v1 B7 v1
Log:

W(A10 v0,A10 v1)
W(B7 v0, B7 v1)

(Log records contain both old and new copies)

Suppose crash, come back, and find disk has A10 v1, B7 v0
Can use the log to go back to both being version 0 or both being version 1

6.033 Computer System Engineering
Spring 2009

6.033 Lecture 17 -- Logging

Sam Madden

Last time: introduced atomicity

two key ideas:
- all or nothing
- isolation

saw transactions, which are a powerful way to ensure atomicity

```
begin
    xfer(A,B,10)
    debit(B,10)
commit (or abort)
```

started talking about

all-or-nothing without isolation
(isolation next time)

saw idea of shadow copies, where you write changes into a secondary copy, and then atomically install that copy

(show slide)

this gets at the key idea of atomicity -- the golden rule

never overwrite the only copy

most atomicity schemes work kind of like this -- you make changes to some separate copy of the data, and then have a way to switch over to that new copy when needed.

if you crash mid-way through making changes, old copy is still there. typically when you crash, because transaction was incomplete, you just revert to old copy.

shadow copy approach has some limitations; what are they?

- only works for one file at a time (might be able to fix using, e.g., shadow directories)

- requires us to make a whole copy of the file

shadow copies are used in many places -- e.g., text editors , (emacs)

today we are going to learn about a general method for all-or-nothing atomicity that addresses these limitations -- **logging**.

basic idea is that after every change, you record the before and after value of the object you changed.

let's work with bank account balances; suppose we have the following table in memory

account id balance
A 100
B 50

suppose I do
    begin
    debit (A,10)
    debit (B, 20)
    end

    begin
    deposit (A,50)

what kinds of things do I keep in the log:
    transaction begin / commit / abort
        transaction id
    updates
        tid
        variable updated
        before (undo) / after (redo) state

log:

    begin 1
    update 1 A before: 100; after: 90
    update 1 A before: 50; after: 30
    commit 1

    begin 2
    update 2 A before: 90; after 140
    crash!

this log now records everything I need to determine the value of an object

```
read(var, log):
        cset = {}
        for r in log(len-1) ... log(0)
                if r is commit
                        cset = cset U r.TID
                if r is update
                        if (r.TID in cset and r.var = var)
                                return r.after
```

but this is really slow, since i have to scan the log after every read

what is the commit point? (when commit record goes to log)

(writes, however, are fast, since i just append to the log, probably sequentially)

how to make this faster? keep two copies -- "cell storage" -- e.g., the actual table contents, in addition to log on disk. reads can just read current value from cell storage, and writes can go to both places.

```
read(var)
        return read(cell-storage,var)

write(TID,var, newval)
        append(log,TID, var, cell-storage(var), newval)
        write(cell-storage,var,newval)
```

let's see what happens:

state:

A	100 --> 90 --> 140
B	50 --> 30 -> 60 -> 30 -> 40

begin	log:
debit (A,10)	begin T1
debit (B, 20)	update(A, T1, 100, 90)
commit	update(B, T1, 50, 30)
	commit T1
begin	begin T2

deposit (B,30)	update(B,T2, 30, 60)
abort	abort T2
begin	begin T3
debit (B,20)	update(T3,B, 30, 40)
commit	commit T3
begin	begin T4
deposit (A,50)	update(T4,A,90,140)
crash	abort T4

After crash, cell storage has the wrong value for A in it. What should we do?

Need to add a recover procedure, and need to do a backwards pass on the log to undo the effects of uncommitted transactions. (undo the "losers")

Anything else we need to do?

Also need to do a forward pass on the log, to redo the effects of cell storage writes in case we crashed before writing cell storage but after doing append. (redo the "winners")

```
recover(log):
    cset = {}
    for each record r in log[len(log)-1 ... 0]  //UNDO
        if r is commit, cset = cset U r.TID
        if r is update and r.TID not in cset:
            write(cell-storage, r.var, r.before)
    for each record r in log[0...len(log)-1]  //REDO
        if r is update and r.TID in cset:
            write(cell-storage, r.var, r.after)
```

Why backwards pass for UNDO? Have to do a scan to determine cset. If we do this scan backwards, we can combine with UNDO.

Why forwards pass to REDO? Want the cell-storage to show the results of the most recent committed transaction.

other variants possible -- e.g., redo then undo, forward pass for undo (with previous scan to determine c set), etc.

what if I crash during recovery?

OK -- recovery is idempotent

example: after crash: A=140; B=40

cset = {1, 3}

UNDO

A -> 90

B -> 30

REDO

A -> 90

B -> 30

B -> 40

T1 and T3 committed, and in that order.

Optimization 1: "Don't REDO updates already in cell storage"

Possible to optimize this somewhat by not REDO updates already reflected in cell storage. Simplest way to do this is to record on each cell a log record ID of the most recent log record reflected in the cell. Diagram:

UPDATE(tid,logid,var,before,after)

(in our example, all we needed to actually do was UNDO A->90)

Q: Why did I write the log record before updating the cell storage? What would have happened if I had done these in opposite order?

cell storage would have value but might not be in log

recovery wouldn't properly undo effects of updates

This is called "write ahead logging" -- always write the log before you update the data

Optimization 2:

Defer cell storage updates

Can keep a cache of cell storage in memory, and flush whenever we want. Doesn't

really matter, as long as log record is written first, since logging will ensure that we REDO any cell storage updates for committed transactions.

```
read(var):
        if var in cache
                return cache[var]
        else
                val = read(cell-storage, var)
                add (var,val) to cache
                return val

write(var,val)
        append(log,TID, var, cell-storage(var), newval)
        add (var,val) to cache
```

Optimization 3:

Truncate the log.

If log grows forever, recovery takes a long time, and disk space is wasted.

Q: What prefix of the log can be discarded?

A: Any part about completed transactions whose changes are definitely reflected in cell storage (because we will never need to reapply those changes.)

Idea:

checkpoint:
write checkpoint record
flush cell storage
truncate log prior to checkpoint

Write checkpoint.
Write any outstanding updates to cell storage to disk. This state written to disk definitely reflects all updates to log records prior to the checkpoint. Truncate the log prior to the checkpoint.

Most databases implement truncation by segmenting the log into a number of files that are chained together. Truncation involves deleting files that are no longer used.

Diagram:

Logging --

- good read performance -- cell storage, plus in memory cache
- decent write performance --
  - have to write log, but can write sequentially;
  - cell storage written lazily

- recovery is fast -- only read non-truncated part of log

Limitations --

- external actions -- "dispense money", "fire missile"
  - cannot make them atomic with disk updates

- writing everything twice -- slower?

Next time --

- isolation of multiple concurrent transactions

6.033 Computer System Engineering
Spring 2009

Lecture 18 -- Isolation + Concurrent Operations

Sam Madden

Key Ideas:

Serial equivalence
Two-phase locking
Deadlock

So far, we've imagined only one transaction at a time -- in effect, a big lock around the whole system (database, file system, etc.).

What's wrong with that? Why does one transaction at a time utilize the system less than running everything serially? (Might expect the opposite to be true, since switching between transactions presumably has some overhead!)

- Hard to exploit multiple processors
- Might want to perform processing on behalf of one transaction while another waits for disk or CPU

Our goal in isolation is serial equivalence -- want final outcome of transactions to be same as running transactions one after another.

A = 50
B = 10

xferPercent(A,B,.2) xferPercent(A,B,.2)

A = 40
B = 20

A = 32
B = 28

xfer(A,B,10)
RA // t = .1A
WA // A = A - t
RB
WB // B = B + t

Example "schedule":

```
RA
WA  // a= 40
        RA
        WA // a= 32
RB
WB // B = 20
        RB
        WB // B = 28
```

Serializable? Yes. Why?
Outcome is the same.

```
RA
        RA
WA // a= 40
        WA  //  a = 40
RB
WB //B = 20
        RB
        WB // B = 30
```

Not serializable!

Is there some test for serializability?

What went wrong in this second example?

T1's read of A preceded T2's write of A, and T2's read of A preceded T1's write of A.

To formalize, define "conflict" -- two R/W operations o1 in T1 and o2 in T2 conflict if either o1 or o2 is a W and both are to the same object.

(aren't worried about two read operations.)

A schedule is serializable if all *conflicting* operations in a pair of transactions T1 and T2 are ordered the same way -- e.g., T1-T2 or T2-T1

Why? Suppose they weren't. Then one transaction might read something before another transaction updated it, and update it afterwards, as in example above. First

transactions effects are lost!

Easy way to test for serializability: conflict graph

Make a graph with nodes as transactions
Draw arrows from T1 to T2 if op1 in T1 conflicts with op2 in T2 and op1 precedes op2

Example:

Now that we understand what it means for a schedule to be serializable, our goal is to come up with a locking protocol that ensures it.

We want to ensure that all conflict operations are ordered in the same way. Use locks to do that. Don't require the programmer to manually get locks.

Instead, transaction system acquires locks as it reads objects.

Locking protocol:
before reading/writing an object, get lock on it
(if lock isn't available, block)

<u>Can I release right away? (No! example):</u>

```
T1              T2
Lock A
                Lock A
        RA      Block
        WA
Release A
                Lock A    //T2 "sneaks in" and makes its updates, violates serializability
                Lock B
                        RA
                        WA
                        RB
                        WB
                Release A
                Release B

Lock B
        RB
```

WB
Release B

Exposed value of B before updating that T2 shouldn't have been able to see -- not serializable.

Locking protocol:
before reading/writing an object, get lock on it
(if lock isn't available, block)

release locks after transaction commit

Lock A	Lock A (block)
RA	|
WA	|
Lock B	|
RB	|
WB	|
commit	|
release A, B	|
	v
	RA
	WA
	Lock B
	RB
	WB

(force T2 to happen completely after T1 -- clearly given up some concurrency; note that if T2 access other objects, e.g., C, first, that wouldn't be a problem.

will address this limitation in a minute.)

Issue: deadlocks

What if T2 tried to get lock on B first?

No transaction could make progress.

Can we just require transactions to always acquire locks in same order? No, because transactions may not know what locks they are going to acquire -- e.g., things that are updated may depend on the accounts that are read (suppose we deduct from all accts with value > x).

So what do we do about deadlocks? Simple: abort one of the transactions and force it to rerun. This is OK because apps must always be prepared for possibility that transaction system crashes and their action aborts.

How do we tell that transactions are deadlocked?

Two basic ways:

1) If a transaction waits more than t seconds for a lock, assume it is deadlocked and restart it.

(Simple, may think a long running transaction a deadlock.) MySQL does this.

2) Build a "waits for graph" -- if T1 is waiting for lock held by T2, draw an edge from T1 to T2. If there is a cycle, then there is a deadlock.

Example:

Making locking protocol more efficient:

Optimization 1:

Once a transaction has acquired all of its locks, it can proceed with out any other transaction interfering with it.

Lock point:
It will be serialized before any other transaction that it conflicts with.

If a transaction is done reading/writing an object, it can release locks on that object *without affecting serialization order*. Doesn't have to wait until the end of the transaction since it isn't going to use the value again.

Example:

```
lock A
        RA                      lock A (block)
        WA
lock B  <--- lock point
release A
                                        RA
                                        WA
                                lock B (block)
        RB
        WB
release B
```

RB
WB

THis is called *two phase locking:*

Phase 1 -- acquire locks, up to lock point
Phase 2 -- release locks, after done with object and after lock point

Optimization 2:

Shared vs exclusive locks

Notice that two read operations of the same object don't conflict -- e.g., if I have two read only transactions, I can interleave their operations however I want and still get the same answer. Protocol didn't allow this, however.

Idea -- have two types of locks: shared (S) locks and exclusive (X) locks. Can have multiple transactions with an S lock, but only one with an X lock.

Transaction can upgrade from S to X if it is the only one with an S lock.

Two phase locking protocol w/ shared locks

Before reading an object, acquire an S lock on it
Before writing an object, acquire an X lock on it

Release locks after lock point

In practice, variants of two phase locking are used:

1) Never release write locks until after transaction commit? Why?

T2 reads A written by T1 before T1 commits -- now T1 aborts, T2 must also abort.

Strict two phase locking

2) Never release any locks until transaction commit? Why?

Can't tell when we are done with locks

Rigorous two phase locking

6.033 Computer System Engineering
Spring 2009

Lec 19 : Nested atomic actions, and multisite atomicity

Sam Madden

So far, we've focused on the case where transactions are running on just one system, and where all of transaction either commits or aborts.

Today -- several additional goals

- Nested atomic actions -- the ability to have sub-actions which may or may not succeed, that don't affect the outcome of the whole action (nested atomic actions)

- Multisite actions -- transactions that span multiple sites (Distributed transactions); why?

    - Performance (2 machines store a database, both machines participate in querying or updating of their part of the DB.)

    - Administrative -- two databases owned by two different organizations, want to run commands that span both

Example: travel site:

Single System:

Wanted nested actions to allow subtransactions to run to completion without committing, and then commit subtransactions

```
Begin  (Parent P)
        Begin  (A)
                Reserve JB
        End
        Begin (B)
                Reserve USAir  ---> if this aborts, JB reservation isn't lost
        End                          ---> if this aborts, P can try something else without
                                                losing JB reservation
End
        Parent P
                Sub xaction A    |  Commit iff
```

Sub xaction B | P commits

Goals:

A + B's effects aren't visible externally unless P commits.
(Once P commits, A + B's effects both visible)

A + B can independently abort

B shouldn't see A's effects unless A has finished.

Once A has reached and, it shouldn't abort unless A aborts (since B may use its results)

We are going say that the a nested transaction that has reached its "End" has "tentatively committed" -- it is ready to commit, but is waiting for outcome of its parent to be decided.

Changes to protocol:
Locking:

When to release A/B's locks? Before P commits? No -- because P might abort!
Only after P commits.

But B should be able to read A's data after it has reached "End".

Change lock acquire protocol to check to see if waiting for a lock from a tentatively committed transactions w/ same parent

Logging:
Need to keep separate begin / end for subactions, in case one of them aborts and we crash, so we are sure its effects are undone.

During log processing, when we encounter a sub-transaction, only want to commit it if its parent commits

Begin P
Begin A
UP seatmap1
Tentative Commit A (Parent P)
Begin B
UP seatmap 2
Tentative Commit B (Parent P)
End P

Recovery: scan backwards, determining winners, undoing losers

(e.g., sub actions that abort or sub actions whose parent aborted)
scan forwards, redoing winners

Multisite Actions

Similar protocol and rules -- suppose now that JB and USAir are separate reservation systems, each with their own data.

There is one coordinator that the user connects to and who makes reservations on these subsystems. Diagram (with internet)

Like before, want JB to commit only if USAir commits, and vice versa.

Harder than before, because now JB / USAir might crash independently, and messages might be lost / reordered.

Separate logs for each site, including coordinator.

To deal with message losses, going to use a protocol like exactly once RPC.

Diagram:

Basic protocol is as follows:

Coordinator sends tasks to workers

Once all tasks are done, coordinator needs to get workers to enter prepared (tentatively committed) state

Tentatively committed here means workers will definitely commit if coordinator tells them to do so; coordinator can **unilaterally commit**

Protocol (with no loss): (prepare, vote, commit, ack)

Why not just send COMMIT messages to all sites once they've finished their share of the work?

One of them might fail during COMMIT processing, which would require us to be able to ABORT suboords that have already committed.

Suppose messages are lost? Use timeouts to resend prepare, commit messages

Crashes? Need to make sure that logs on workers ensure that they can recover into the tentatively committed state.

Log records:

Write TC record on workers before "Yes" vote.

Commit record still written on coordinator -- that is commit point of the whole transaction

Coordinator also writes a "done" message

Suppose worker crashes:

Before prepare?

After prepare?

Suppose coordinator crashes:

Before prepare

After sending some prepares

After writing commit?

After writing done?

How does coordinator respond to "TX?" inquiry? Does it keep state of all xactions forever? (No -- once it has received acks from all workers, it knows they have received outcome.)

Notice that workers *cannot forget state of transaction* until after they hear commit / abort from coordinator, even if they crash. This makes protocol somewhat impractical in cross-organizational settings.

What to do instead?
Use compensating actions (e.g., airlines will allow you to cancel a purchase free of charge within a few hours of making a reservation.)

2PC provides a way for a set of distributed nodes to reach agreement (e.g., commit or abort.) Note, however, that it only guarantees that all nodes eventually learn about outcome, not that they agree at the same instant.

"2 Generals Paradox" (slides)

Can never ensure that agreement happens in bounded time (though it will eventually happen with high probability.)

6.033 Computer System Engineering
Spring 2009

Today: Consistency and Replication

Sam Madden

Exam: Mean 55. If your grade was low -- don't sweat it. What matters is how you performed relative to the mean.

Last few lectures: seen how we can use logging and locking to ensure atomic operation, even in the face of failures.

But -- we might want to do better than just recover from a crash. We might want to be able to mask the failure of a system -- make the system more available.

Previously seen that replication is a powerful tool that help accomplish that. Today we will investigate replication in a bit more detail, and look some problems that arise with maintaining replicated systems.

Replication used in both the local area -- e.g., two replicas on the same rack of a machine room, as well as the wide area -- two replicas in different buildings, states, etc.

Wide area replication important for **disaster recovery.** "Half width of a hurricane."

Let's look at how we might apply replication to a system:

(Send requests to both; if one is down, just send to the other one.)

What could go wrong:

(Network partition. Some other client sends requests to other machine. Now they are out of sync.)

Solutions:

- Eventual consistency -- have some way to sync up after the fact (exposed inconsistency -- this could have never happened if we didn't have a replicated system.)

- Strict consistency -- "single-copy consistency" -- ensure that an external user can't tell the difference between one machine and several machines.

strict consistency and availability fundamentally at odds, esp. in the face of network partitions.

In above example, when the eventual consistency approach allowed either client to receive answers.

In strict consistency approach, best we can do is to designate one replica as the "master" (or authority) and allow clients talking to it to proceed. Other clients -- talking to other replica -- will not get service.

Eventual consistency

Easier to provide (usually) than strict consistency. Typical way that this works is to associate a time stamp (or version number) with each data item.

Have replicas periodically compare time stamps, use freshest. This is what Unison does.

DNS is an example of eventual consistency -- suppose I change:

db.csail.mit.edu from IP X to IP Y

Effect won't be visible immediately; instead, will propagate when the timeout on the DNS record expires. Remember -- DNS servers cache lookups for a fixed amount of time.

(Show slide)

This means that the machine with IP X may receive requests for db.csail.mit.edu for some period of time.

Inconsistency is considered acceptable here. Providing strict consistency here would be very hard: I'd have to contact every cache in the world and ensure they were

updated, or would have to have no caching and have every request come to csail.mit.edu on every lookup. Neither is OK.

One challenge with eventual consistency is what to do if there are concurrent updates to the same data. Not a problem in DNS because there is one master for each subdomain (e.g., csail.mit.edu) that totally orders updates. But in a system like Unison, we can have multiple updates to the same data item, which is problematic and requires some kind of manual conflict resolution.

Will see another example of eventual consistency next recitation, and see some of the additional pitfalls that concurrent updates to the same item can lead to.

Strict Consistency -- Replicated State Machines

Basic idea is to create *n* copies of system. Same starting state. Same inputs, in the same order. Deterministic operations. Ensures the same final state.

If one replica crashes, others have redundant copies of state and can cover for it. When that replica recovers, it can copy its state from the others.

But what about network partitions -- replica may have missed a request, and may not even know it!

If requests come from different clients on different sides of the partition, then replicas can see different requests. Diagram:

One option: create an ordering service that totally orders all requests. When a network partition happens, one replica may miss some requests, but it can detect that

and request those operations from other replicas before proceeding.
Diagram:

(Single point of failure -- will return to that in a minute.)

Degenerate version: Single Master Replication

Diagram:

All updates go to a single master, which is responsible for propagating updates to the replicas.

Seems like it might fix the problem because now there is only one master.

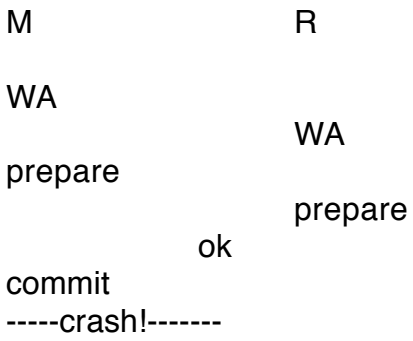
If master fails, switch over to one of the replicas. What's the problem here?

May have been writes which propagated to some of the replicas, but not others. How do replicas know what the current state is?

(They don't -- might think they could talk to each other and figure out the most update any of them had seen. But there might be something the crashed primary saw that they hadn't.)

Does two phase commit solve this problem?

(No. Suppose following



)

One approach is to simply tolerate some inconsistency -- e.g., accept that replicas may not know about the outcome of some transaction.

In practice, this is what is done for wide area replication -- e.g., in a database system.

How does the client know which server to talk to?

Ordering Service

How to build a reliable ordering service?

1 node -- low availability.

Use a replicated state machine. Diagram:

Still have problem with partitions. Typically solution is to accept answer
if a majority of nodes agree on it.
Challenge: network partitions, message re-orderings, dropped messages.

This problem is called agreement:

Agreement protocol (sketch)

Leader:	Worker
next(op)	
	agree(op)
accept(op)	
	accept_ok(op)
commit	

Phase 1: Leader node proposes next op to all others (propose)

Phase 2 : If majority agrees, it tells all others that this is the next op (accept)

Phase 3: Once a majority acknowledge accept,
Send out next op!

Reason for this protocol is we want to allow other nodes to continue if leader goes away.

Other nodes:

If after a random timeout can't contact leader, propose themselves as new leader. If majority agree, they are new leader.

(Nodes only respond to prepare / accept requests from their current leader.)

Once a new leader has been chosen, it checks to see if any node has accepted anything. If not, it restarts protocol.

If some nodes have agreed but no node has seen an accept, doesn't matter, because that transaction must not have committed yet, so leader can safely abort it.

Once a leader has received an accept_ok from from a majority, it knows that even if there is a network partition, where it is in the minority partition, majority partition will have at least one node that has committed.

Otherwise, majority partition might just have one node that is prepared, new node might assume that the transaction didn't commit and propose something else.

Nodes outside of the majority are inconsistent and need to recover, even if they have already sent an accept_ok message.

Commit point is when n/2+1th worker writes accept(ok).

Example 1 -- network partition

Example 2 -- failure of a leader

These agreement protocols are very tricky to get right, which is why I'm only sketching the idea here. Take 6.828 to learn more.

```
Protocol sketch:
        N :  number of replicas (odd, e.g., 3 or 5)
        votes = 0
        voters = {}
        R = 0 //must persist across reboots
        opQ = new Queue()  //must persist across reboots

        do(op):
                opQ.append(op)
                broadcast("next op is opQ[R], in round R");

        periodically:
                if (R < opQ.len)
                        broadcast("next op is opQ[R], in round R ");

        next(op,from, round):
                if (R < round):
                        wait to see if any more votes for R arrive
                        if (R < round):
                                recover
                                return

                if (op == opQ[R] && from not in voters):
                        votes++;
                voters = voters U from


                if (votes > N/2):
                        notifyWorkers (opQ[R])
                        R = R + 1
                        voters = {}
                        votes = 0

                if (N/2 - votes > N - len(voters)
                        deadlock
```

6.033 Computer System Engineering
Spring 2009

Security intro
Nickolai Zeldovich
==========

key ideas for today:
    key to security is understanding what the attacker can do
    principles: reduce trust (least privilege), economy of mechanism

--- board 1 ---

security / protection
    permeates computer system design
    if you don't design it right upfront, can be hard to fix later
    much like naming, network protocols, atomicity, etc
    will affect all of the above

give examples of security problems
    SLIDE: general security stats
      critical problems that allow attackers to take control over
      windows machines -- about once a week
    SLIDE: not surprising, then, that china controls computers in embassies
    SLIDE: even medical devices like pacemakers are vulnerable to attacks

so what is protection? back to first board
    prevent access by bad guys
    allow access by good guys
    policies [lots of them, and we can't really cover all possibilities]
    [.. so much like with other topics] -> mechanism

--- board 2 ---

real world vs computer security
    same:
      lock - encryption
      checkpoints - access control
      laws, punishment
    different:
      attacks are fast, cheap, scalable
          ~same effort to compromise 1 or 1,000,000 machines
          can compromise machines all over the world
          no need to physically be present at the machine
          no strong notion of identity
      global; few laws

--- board 3 ---

policy goals: positive vs negative
    Nickolai can read grades.txt -- easy
      why easy? build system, if it doesn't work, keep fixing until it does
    John cannot read grades.txt -- hard
      seems just the opposite of the above
      we can try asking John to log in and try to access grades.txt
      not enough: have to quantify all possible ways John might get grades.txt
          tries to access the disk storing the file directly
          tries to access it via a browser (maybe web server has access?)
          tries to read uninitialized memory after Nickolai's editor exits

tries to intercept NFS packets that are reading/writing grades.txt

tries to sell you a malicious copy of Windows
tries to take the physical disk out of server and copy it
tries to steal a copy of printout from the trash
calls the system administrator and pretends to be Nickolai

hard to say "regardless of what happens, John will not get grades.txt"

not enough to control access via one interface
must ensure all possible access paths are secure

we've seen some positive goals (e.g. naming) in 6.033 already
some negative goals too (transaction must not be "corrupted")
security is harder because attacker can do many things
with transactions, we knew what's going to happen (crash at any point)
most security problems are such negative goals

--- board 4 ---

threat model
the most important thing is to understand what your attacker can do
then you can design your system to defend against these things

C -> I -> S

typical setting:
client named Alice, server named Bob
an attacker (router) in the network, Eve, is eavesdropping
alternatively, Lucifer, a malicious adversary, can send, modify packets

does attacker control the client? server?

frequent assumption:
no physical, social engineering attacks
only intercept/send messages
might or might not compromise server, client

this picture applies even on a single machine
processes from diff. users making calls into the OS kernel

consider costs as well (both security and insecurity have a price)
convenience, HW expense, design, ..

right side of the board:
basic goals
- authentication [SLIDE: kentucky fax]
- authorization [who is authorized to release prisoners?]
- confidentiality NOTE: quite diff. from authentication!
- auditing
- availability

--- board 5 ---

policies / mechanisms
hardware: confine user code
mechanism: virtual memory, supervisor bit

```
      authentication: kernel initializes page table, supervisor bit
                  HW knows current state
      authorization: can access any virtual memory address in current PT
                   cannot access privileged CPU state
    Unix: private files
      mechanism: processes, user IDs, file permissions
      authentication: user password, when user starts a process
      authorization: kernel checks file permissions
    firewalls: restrict connections
      mechanism: packet filtering
      authentication: connection establishment
      authorization: list of allowed/prohibited connections
      seemingly weak mechanism, but surprisingly powerful in practice
    bank ATM: can only withdraw from your acct, up to balance
      mechanism: app-level checks in server process
      authentication: card & PIN
      authorization: track account balance
    cryptography: next lectures

--- board 6 ---

challenges
    bugs
      hard to build bug-free systems, write perfect code
      expect bugs, try to design your system to be secure despite them
      in recitation tomorrow, will look at some of these bugs
    complete mediation
      requires careful design
      SLIDE: paymaxx bug
    many mechanisms: hard to enforce coherent policy
      want to ensure that bank policies are followed
      what mechanisms do we have?
          virtual memory isolates processes
          kernel, file system implements ACLs
          bank ATM implements its own checks
          web banking might implement other checks
          system used by bank employees has other checks
          firewalls at different places in the network
    interactions between layers
      [caching/timing, naming, memory reuse, network replay]
      SLIDE: naming problem with symlink attacks
      SLIDE: password checking one character at a time

--- board 7 ---

safety net approach
    be paranoid -- make assumptions explicit
      attackers will try all possible corner cases
    consider the environment
      if you are relying on network security, check for open wireless networks
      if you are reusing, relying on another component, make sure it's secure
          code meant to run on non-networked system used on the web?
          never expected to deal with malicious inputs
    consider dynamics of use
      suppose only Nickolai should access grades.txt
          who can specify permissions for the grades file?
```

```
          who can modify editor on Athena?  or set permissions on it?
          who can control name translation for that file?
    defend in depth
      even if you have a server on a "secure" company network, still want
      to require passwords.  what if someone brings an infected laptop?

right side of the board:
    humans: weakest link
      - UI
      - safe defaults

--- board 8 ---

design principles
    open design, minimize secrets
      figure out what's going to differentiate bad guys vs good guys
      focus on protecting that, make everything else public
      authentication: ID public, sth. that proves you're that ID is secret
      SSNs, credit card numbers fail at this
          SSNs used both as ID and as credentials for authentication
          unclear what part of credit card number is really secret
          some receipts star-out first 12 digits, other star out last 4
    economy of mechanism
      simple security mechanism
      multiple security mechanisms interfere
      try hard to reduce security policies to existing mechanisms

      design to minimize "impedance mismatch" between security mechanisms
          usually a number of app layers between client and real object
          right side: diagram: Client-WebApp-FS-Disk
          suppose this is paymaxx which stores user tax data
          would be great if policy were enforced on obj directly
            then wouldn't have to trust the server app code
          suppose Obj is file -- mechanism is file permissions
            if diff users store their data in 1 file, can't use OS prot
            if we carefully design files 1 per user, may be able to use OS

    least privilege: minimize TCB
      TCB (trusted computing base)
      usually don't want to trust the network (next lectures will show how)
      break up your app into small components, each with least needed
privilege
```

6.033 Computer System Engineering
Spring 2009

Nickolai Zeldovich

key ideas for today:
    open design
    identity vs authenticator
    authenticating messages vs principals (message integrity, bind data)
    public key authentication

--- new board ---

security model
    C - I - S
    last time:
      talked about some general ideas for how to build secure systems
      defensive design: expect compromise, break into parts, reduce privilege
      last recitation, saw examples of what can cause parts to be compromised
    for the rest of the lectures:
      assume that we can design end-points to be correct & secure
          (hard but let's go along with this for now)
      figure out how to achieve security in the face of attackers
          attackers can look at, modify, and send messages
    basic goals that we want to achieve
      inside the server: guard - service

      authentication
      authorization
      confidentiality

--- new board ---

basic building block: crypto
    let's look at how you might implement encryption
    two functions, Encrypt and Decrypt

    C -> E -> I -> D -> S

    military systems, E and D are secret
    closed design

    problem: if someone steals your design, you're in big trouble
    hard to analyze system without at the same time losing secrecy
    key principle in building secure systems: minimize secrets!

--- new board ---

open design
    big advantage: if someone steals design & key, can just change keys
    can analyze system separately from the specific secret key
    minimizes the secrets

    important principle in designing systems:
      figure out precisely what secrets distinguish bad guys from good guys
      it's very hard to keep things secret
      knowing what's important will allow you to focus on the right things

    same diagram but with keys going into E & D

example of symmetric key crypto: one-time pad
    XOR the message with random bits, which are the key
    quickly describe XOR, why you get the original message back
    problem: key is giant (but scheme is perfectly secure)

stream ciphers: various algorithms that generate random-looking bits
    no longer perfectly unbreakable, just requires lots of computation
    SLIDE: RC4

attack if keys reused
    C->S: Encrypt(k, "Credit card NNN")
    S->C: Encrypt(k, "Thank you, ...")

    XOR two ciphertexts and known response to get unknown request message!
    never reuse keys with symmetric crypto! (one-time pad!)

--- new board ---

previously needed shared keys, doesn't scale

RSA: public-key cryptography
    keys for encryption, decryption differ
    SLIDE: RSA algorithm
      short example computation?

      p = 31, q = 23, N = 713
      e = 7, d = 283

      m = 5
      c = m^e mod N = 5^7 mod 713 = 408
      m = c^d mod N = 408^283 mod 713 = 5
    difficult to generate e from d, and vice-versa
    assumption: factoring N is hard!

    much more computationally expensive than symmetric-key crypto!
    important property: don't need a shared key between each party
      encrypting a message for someone is diff. than decrypting it
      server can use the same key for many clients sending to it

    similarly tricky to use in practice
      how to represent messages?
      small messages are weak
      large messages are inefficient
      can multiply messages together
      need something called padding

crypto mechanisms rely on computational complexity
    pick key sizes appropriately -- "window of validity"

--- new board ---

principal authentication
    principal/identity: a way of saying who you are
    authenticator: something that convinces others of your identity
      open design principle sort-of applies here

```
      want to keep identity public, authenticator private
      focus on what's distinguishing good guy from bad guy
    usually there's a rendezvous to agree on an acceptable authenticator

authenticator types: right side of the board
    real world: SSN
      bad design: confuses principal's identity and authenticator
    passwords
      assuming user is the only one that knows password, can infer that
          if someone knows the password, it must be the user
      server stores list of passwords, which is a disaster if compromised
      common solution: store hashes of passwords
          define a cryptographic hash:
            H(m) -> v, v short (e.g. 256 bits)
            given H(m), hard to find m' such that H(m') = H(m)
          foils the timing attack we had last time
          in theory hard to reverse
          dictionary attack: try short character sequences, words
    physical object
      magnetic card: stores a long password, not very interesting
      smartcard: computer that authenticates using crypto
    biometric
      oldest form of authentication: people remember faces, voices
      can be easy to steal (you leave fingerprints, face images everywhere)
      unlike a password, hard to change if compromised
      more of an identity than authentication mechanism

    need to trust/authenticate who you're providing your authenticator to!
      fake login screen, fake ATM machine can get a user's password/PIN
      next recitation you'll read more about what happens in the real world
      web phishing attacks: convincing you to authenticate to them

--- new board ---

suppose we trust our client (e.g. laptop, smartcard, ...)
how to design protocol?
    board: C - I - S diagram
      client sending a message saying "buy 10 shares of Google stock"
    simple version: just send password over the network
      attacker has password, can now impersonate user
    better version?  send a hash of a password
      attacker doesn't get our password (good, probably)
      but the hash is now just as good -- can splice it onto other msg!

    ** need both authentication AND integrity **

    better?  include checksum of message, eg CRC
      attacker can re-compute checksum!  need checksum to be keyed

    better yet: send a hash of [ message + password ], called a MAC
      message authentication code
      if you're going to do this: look up HMAC
    best: establish a session key, minimize use of password (long-term secret)
      send a message to the other party saying "i will use this key for a bit"
      use that key to MAC individual messages
```

6.033 Computer System Engineering
Spring 2009

Nickolai Zeldovich

high-level ideas for today:
    public key authentication
    certificate authorities
    session keys
    network protocols and attacks: be explicit

Authentication, Authorization
==========

--- board 1 ---

setting: client and server sending messages over the internet
    want to achieve: authentication, authorization, confidentiality

last time: how to authenticate requests given a shared secret
    stock trading example: "buy 100 shares of google"
    password, hash the request and password together

```
 [pw]           [pw]
    C --- I --- S

     <----------
     m, H(m + pw)   m: "GOOG shares cost $100"

     ---------->
     m, H(m + pw)   m: "buy 100 shares of GOOG"
```

--- board 2 ---

now: what to do if we don't have a shared secret ahead of time?
    e.g. stock quotes coming from web server

    want a pair of functions SIGN and VERIFY

```
                       Shared           Public
                        key           key

      SIGN(m, K1) -> sig      H(m+K1)           m^e mod N
      VERIFY(m, sig, K2)      sig==H(m+K2)      sig^d mod N == m

                       MAC         Signature
```

    SLIDE: RSA reminder

      TELL STUDENTS that both of these examples are crippled
          want to use HMAC and better public-key signature schemes
          these are just for your intuition

    formalize previous password-based authentication: MAC
    new: different sign, verify keys
    key property of public-key signatures: others can verify but not generate!

attacks on SIGN, VERIFY for authentication [right side of board?]
    crypto attacks

```
      figure out the key or find ways to violate crypto's guarantees
    authentication
      modification
      reordering
      extending
      splicing

    for example RSA as-described is vulnerable to multiplying messages
      also need to take care to condense string into an integer

--- board 3 ---

what does it even mean to authenticate someone in this case?
    how do you know what public key to use for someone?
    idea: split up into authenticating a name, and then trusting a name
    someone is going to manage names for us

    symmetric
      S <-> PW <-> trust

    asymmetric
      S <-> K1 <-> name <-> trust
            \----V----/
            trust someone to tell us these bindings

--- board 4 ---

talk to a server, get a response and signature with server's key
    now talk to a key-naming server to ask it, what's this guy's name?
    key-naming server responds, along with its own signature
    must trust key-naming server ahead of time
    otherwise might need to talk to another key-naming server...

      SIGN("GOOG is $100", K1), K2

    [C knows K2_ca]

      C <--------- S
      ^
      |
      \----> CA
      "Who is K2?"
      SIGN("K2 is quotes.nasdaq.com", K1_ca)

--- board 5 ---

idea: don't actually need to query the key-naming server all the time
    give key-naming server's signed response to original server
    typically called a "certificate"

 [K2_ca]
    C <--------------- S

      m: "GOOG is $100"
      SIGN(m, K1_q), K2_q
      SIGN("K2_q is quotes.nasdaq.com", K1_ca)
```

delegation: certificates might be chained
      otherwise everyone has to talk to the root certificate authority
    this is roughly how DNSSEC works
    user trusts the top-level certificate

      SIGN("K2_q is quotes.nasdaq.com", K1_nasdaq), K2_nasdaq
      SIGN("K2_nasdaq is *.nasdaq.com", K1_ca)

THE ENTIRE CHAIN HAS TO BE TRUSTED!
    and infact the weakest possible way to construct the chain is the problem

how would one of these CAs decide who the next guy is?
    used to be that you have to go to a town office and get a business license
      can specify any name you want, can probably find some town that's lax
    now it's just verified by email on your domain registration
      one popular CA will send email to webmaster@domain.com to ask if it's OK

SLIDE: easy to spoof this process, even for well-known companies!
    reiterate: weakest link in the chain is what matters

SLIDE: rarely-used mechanisms are a disaster for security
    Verisign designed a certificate-revocation list for this purpose
    not part of the usual workflow, so never tested!
    indeed, when time came to use it, noone knew how to find this CRL!

in practice users would think that a secure website was secure, end of story
    didn't bother thinking about what server they were actually talking to
    new kind of certificates now that involve "extended validation"
    pops up a green bar showing the name of the company that was validated

alternative: blindly trust & remember keys
    vulnerable first time around
    SSH
    add a certificate exception in a browser

--- board 6 ---

ok, now have public keys for both parties, what next?
    asymmetric encryption/signatures are computationally-expensive
    most protocols use public keys to establish symmetric session key
      then use symmetric encryption or MACs for subsequent messages

session key protocol: denning-sacco
    A wants to talk with B, need to establish symmetric session key
    1. A -> S: A, B
    2. S -> A: SIGN({A, Ka}, Ks), SIGN({B, Kb}, Ks)
    3. A picks a session key Kab
      A -> B: SIGN({A, Ka}, Ks), ENCRYPT(SIGN(Kab, Ka), Kb)
    4. A and B communicate using Kab to do symmetric encryption/MACs
      (usually not a good idea to use same key for two purposes, so
       maybe Kab is actually two keys, one for encrypt, one for MAC)

--- board 7 --- [to the right?]

desired goals

1. secrecy: only A, B know Kab
2. authentication: A, B know other's identity

is this true?
  1. A knows Kab, and the only person that can decrypt msg 3 is B
  2. A knows the only person that can decrypt Kab must be B
     B knows the only person that could have sent Kab was A (signed!)

turns out the protocol is broken!
recipient can't really infer the party on the other end is A!
suppose B was malicious (e.g. A visited a malicious web site)
can impersonate A to some other party (e.g. another web site)
  .. get {C:Kc}_Ks
  3. B -> C:
      SIGN({A, Ka}, Ks), ENCRYPT(SIGN(Kab, Ka), Kc)
what recipient can infer is that A signed this message, but so what?
  it wanted to use Kab to talk to someone, but not necessarily us (rcpt)!

6.033 Computer System Engineering
Spring 2009

Nicolai Zeldovich

high-level ideas for today:
    attacks on network protocols
    principles for building secure protocols
    secure comm. channel abstraction; encryption and MAC
    authorization: lists vs tickets
    how to put it all together (security in HTTP)

Protocol Security

SLIDE: reminder of Denning-Sacco protocol

--- board 1 ---

protocol-level attacks
    impersonation: passwords are particularly bad [if used on multiple sites]
    replay
    reflection

SLIDE: reminder of how Denning-Sacco is broken
SLIDE: how to fix Denning-Sacco

protocol goals [right side of the board]
    appropriateness [explicitness?]
      explicit context, name principals
    forward secrecy [compromised keys do not reveal past communication]
      session keys, version numbers
    freshness [distinguish old and new messages]
      [later:] timestamp, nonce

--- board 2 ---

replay attack example:
    C -> S: "Buy 100 shares of GOOG", HMAC(m, pw)
    attacker retransmits the message many times!

    what's the fix? include a nonce, sequence number, timestamp, ..

--- board 3 ---

reflection attack example
    suppose Alice and Bob share a secret (e.g. password)
    want to ensure that the other party is present / alive
    shared key K between A & B

    A -> B: ENCRYPT(Na, K)
    B -> A: Na

    B -> A: ENCRYPT(Nb, K)
    A -> B: Nb

    attack: Lucifer pretends to be Alice, whereas Alice is really disconnected
      Bob asks Lucifer to decrypt challenge using their shared key
      Lucifer doesn't have the key, but can ask Bob to prove himself to Alice

Bob will decrypt his own challenge and send it to Lucifer!

--- board 4 ---

hard to reason about all these individual things when building a system
these building blocks are too low-level
common solution: abstraction of a secure communication channel

C <===> S

provides either confidentiality & authentication, or just authentication

+ long-lived comm
- short-lived sessions, many nodes
- offline messages, outside the model [e.g. confidentiality-only]

for -: have to use the lower-level abstractions we've been talking about

plan: use some protocol like denning-sacco to establish session key
or two keys, one for encryption, one for MACs
use session key for symmetric crypto for secure comm channel

need to include sequence numbers in messages to avoid replay, splicing

example: SSL/TLS in browsers
SLIDE: overview of SSL/TLS from wikipedia

[[encryption attacks:
ciphertext-only
known-plaintext
chosen-plaintext
chosen-ciphertext
]]

SLIDE: just for fun, watch what's happening on the wireless network
explain what tcpdump does

SLIDE: explain what we get out of it: packet headers, etc

SLIDE: gmail not encrypted

SLIDE: facebook not encrypted; plaintext

why don't people use SSL everywhere?
they probably should
some technical reasons: performance
bandwidth, CPU to encrypt messages is negligible now
but SSL requires handshake: 2 more RTTs

--- board 5 ---

secure comm channel =/= security
lots of trust assumptions underlying security of comm channel
claim: you're sending/receiving data to/from a particular entity
who is the entity? someone that a CA verified as being that entity
show amazon.com's certificate

```
      who are all of the CAs?  show list of all CAs in a browser?
          Edit -> Prefs -> Security -> List Certificates
      where do these CAs come from?  comes with browser from Mozilla or MS
    what can go wrong?
      maybe my trust in the name is misplaced (confusing name?  amaz0n.com?)
      maybe CA failed to properly verify the name?
      maybe one of the CAs was compromised?
      maybe one of the CA's keys was disclosed, stolen, guessed?
      your laptop was compromised, someone added an extra trusted CA root?
          alternatively, just modified your browser to do arbitrary things
    higher-level problems
      credentials leaked
      incorrect access control

--- board 6 ---

authorization
    lifetime of authorization:
      1. authorization
      2. mediation
      3. revocation

    revocation --\
    authorize -\ |
               V V
    client -> guard -> server -> object
    guard checks whether request should be allowed

--- board 7 ---

how does this guard work?
    logically, an access matrix

        F1 F2 F3
    A   R  R  RW
    B   RW RW --
    C   -- RW R

    principals on one axis -- need authentication
    objects on another axis -- guard needs to understand what's going on
    entries are allowed operations

where does this matrix get stored?  [right side of board]
    lists (ACLs in Unix, AFS): store column with the object
        authentication: figure out who the user is
      authorization: add user to ACL
        mediation: look up that user's entry in the object's column
    tickets (Java ptr, some URL, some cookies, Kerberos, certs): user stores
row
        must ensure the user can't tamper with his own row!
        authorization: out of scope -- however the user got the ticket (eg.
pw)
        mediation: look up current object in the user's supplied row

--- board 8 ---
```

```
    advantage: ticket model allows delegating privileges easily
    advantage: decouples authorization check from authentication check
        can change the two parts of the system separately
    disadvantage: need to get ticket first
    disadvantage: revocation is difficult
        plan A: chase down every ticket with each user
        plan B: invalidate all outstanding tickets, require users that are
            still authorized to get new tickets

--- board 9 ---

good ways to generate ticket?
    HMAC is easy
    [ resource/rights, gen#/timeout?, ... ], HMAC'ed with server key
    gen# supports revocation
    timeout similarly helps control ticket distribution [Kerberos]

    so let's see how this would work in practice
      ticket to access directory /mit/6.01
      expires 11/11/2009

      ticket = { /mit/6.01, 11/11/2009, MAC(/mit/6.01 + 11/11/2009) }
      ticket = { /mit/6.011, 1/11/2009, MAC(/mit/6.011 + 1/11/2009) }

      CRUCIAL: ensure all signed messages are unambiguously marshalled!

capabilities = naming + tickets
    combine names and tickets together
    can't talk about an object without also talking about the rights to it
    would have avoided the symlink attack

    file descriptors in Unix are sort-of like this

managing this matrix can be hard (either rows or columns)
    common simplification: roles or groups

--- board 10 ---

how are all these components put together?
    example: web browser/server interaction
    secure comm channel: SSL
    client (browser) authenticates the server's name
    server authenticates client's certificate (MIT personal certificates)
    otherwise server gets a connection to someone that hasn't authenticated
yet

    message sequence
      server: identify yourself           \
      client: username, password           > SSL (usually)
      server: here's a cookie (ticket)    /
      client: request 1 and cookie (ticket)
      client: request 2 and cookie (ticket)
      ...

    SLIDE: plaintext cookies: enough to log into google calendar
    SLIDE: plaintext cookies 2: list of other sites
```

often these cookies are valid even after you change your password!
because it's just a signed ticket stating your username
good idea: include pw gen# in cookie/ticket

tickets and ACLs not as different as they sound
tickets often issued based on underlying ACLs
ACLs often used to decide higher-level ops based on lower-level tickets
e.g. server certificate = ticket for server's principal
but other checks apply to server's name
client cert/cookie = ticket for that user's client privileges
but other checks apply to client's name