LEC2: Naming

- motivations
    - build upon the telephone and central directory model, which both can’t scale and aren’t reliable
- namespace, set of values, translation algorithm
    - consider: internal structure of names, can a single name translate to multiple values?
    - DNS: hostname -> IP addr
    - absolute (Trailing dot) vs relative path names
- DNS alg
    - name hierarchy
    - send a request to a root name server or any name server who tries to resolve it or respond w a referral
        - the server that holds either a name record of a referral record for a domain name is known as the **authoritative name server** for a name or name domain.
    - name servers all have power over certain set of names
        - 1 zone can have multiple name servers
    - recursive queries - server holds responsibility for resolving the request itself, and then caches everything its heard about
    - caching
- synonyms -> indirect names, like if you want to rename something, you can just bind the indirect (new) name as the name of the website or something

LEC3: Virtualization

- Idea: virtualize memory so multiple programs can use 1 CPU
    - each program has its own virtual address space
    - communicate via bounded buffer
- VM -> MMU -> Physical addresses

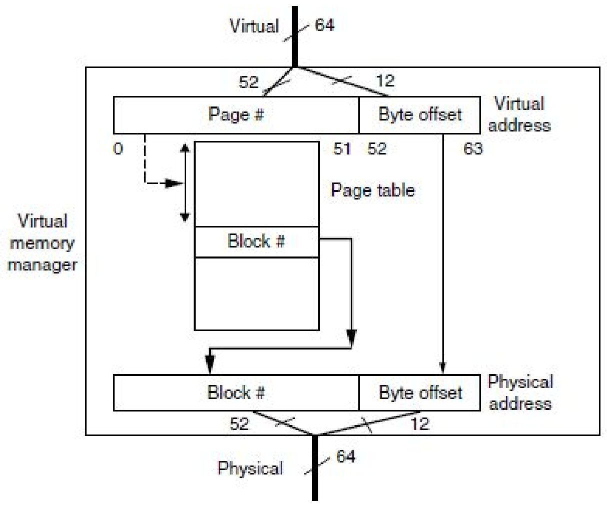


FIGURE 5.19
An implementation of a virtual memory manager using a page table.

-
- VA = page number + offset (page # in page table)
    - page table has other stuff too, like present bit, R/W bit, U/S bit
    - bad point is that this has multiple lookups
- kernel - protected trusted intermediary that can execute some privileged instructions
    - VM doesn't protect the page table so you have to make sure you switch b/w kernel and user mode via interrupts
- multiple threads also use multiple domains via VMM

REC3-4: UNIX

- file descriptor - small integer to identify the file in subsequent calls to r/w
- file system implementation
    - i-number
    - i-list - i# indexed into <-
    - to find i-node, which has file metadata
    - space on disk is split into 512-byte blocks, so the i-node contains addresses of blocks that belong to it.
- image - computer exec environment
- process - execution of an image
    - fork splits an existing process into two independently executing processes
- standard I/O - 1 is open for writing, 0 is open for reading, 2 is associated with the terminal output stream

LEC4: Bounded Buffers/Concurrency/Locks

- for programs to communicate, use **bounded buffers to virtualize memory to allow programs to communicate**
- bounded buffer: a buffer to store up to N messages
  - API: send(m) or receive()
  - send(bb, message):if theres room in the bb, then put the message into the buffer
  - receive(Bb): if theres something in the buffer, take it out
- locks - acquire, release
- Bounded buffers must use locks to write or send messages
  - some problems we encounter with deciding how to create atomic actions is performance, correctness, deadlock

LEC5: Threads

- thread = virtual processor, suspend() - save all registers, memory into memory, resume()
- yield() = suspend the running thread, and choose a new thread to run (via round robin). resume the new thread
- condition variables: let threads wait for events, and get notified when they occur
  - wait(cv, lock) - make sure to release the lock
  - notify(cv) - iterate thru all threads to find the thread w/ cv as 'cv' and continue running it
- preemption: forces a thread to be interrupted so that we dont have to rely on programmers using yield()
  - so that threads will all yield and wait at some point
  - solution to prevent deadlock is a hardware mechanism to disable interrupts

LEC6: Operating Systems

- what do we want out of an operating system? we want to enforce modularity
  - programs shouldn't be able to corrupt each others' **memory** (**VM**)
  - programs should be able to **communicate** (**bounded buffers**)
  - programs should be able to **share** a CPU (**threads**)
- Multiple VMs running on the same hardware -> both have guest OSs that run instructions directly on CPU, where there is also a VMM to restrict privileged instructions
- VM memory
  - guest virtual -> guest physical -> host physical
  - when guest OS tries to load in the PTR, the VMM intercepts, and uses the guest OS page table and VMM page table to map to a host physical address
  - modern hardware has physical hardware doing both table things

- U/K bit
- monolithic kernel - no enforced modularity within the kernel itself
    - so why are there so many bugs?
        - well the thing is complex
- microkernel: enforce modularity by putting subsystems in user programs
    - so like theres one for network, one for device driver
    - but theres high communication cost between modules
    - redesigning monolithic kernels as microkernels is challenging

REC5: Eraser - quick tldr

- tryna detect dynamically race conditions via lockset alg, which is narrowing down the set of possible locks until you get to one, or none!
    - introduces states for each dynamically allocated variable to allow threads to read only, initialize, etc
- might have some false alarms - they label for those
    - memory reuse
    - private locks
    - benign races
- some things they wanted to do were simplicity, and its generally scalable, but is it correct? eh

LEC7: Performance - Question: **How do we get OS's to work well?**

Well we could buy new hardware. But not all aspects improve at the same pace, and Moore's Law is plateauing....

<u>Consider</u>:
Improving Performance in 2 easy steps
1. identify bottleneck
2. relax bottleneck

Some metrics for bottleneck: latency is the time required to perform some action. Throughput is the # of actions executed in some unit of time.

<u>Observe:</u>
- Low # of users
    - low latency, low throughput
    - Doesn't take a lot of time to do stuff, and not many people are asking for things

- Moderate #
    - Low latency, high throughput
    - Still doesnt take that long, but more people are asking for things
- High #
    - High latency, plateau’d throughput
    - Requests are queuing up and it can’t serve requests any faster after a pt

<u>Revisit above def:</u> Instead of identifying the bottleneck, **compare system to our system model to find bottleneck.**

<u>How to relax the bottleneck:</u>

- better algs.... or concurrency it to improve latency/throughput
- batching, caching, concurrency, scheduling, etc

*1) Let’s think about disk throughput... like **r/w speed***

- HDD - platters on a rotating axle -> tracks -> sectors
    - disk head on arm r/w sectors as they rotate past
    - EX. if a rotational speed is 7200 RPM, then it can be like 8.3ms per revolution, and avg read seek is 8.2ms, avg write seek is 9.2ms
- SSD - organized into cells -> pages -> blocks. r/w happens at page-level.
    - erases, overwrites are block-level. It takes a high voltage to erase
- so a thought would be that random accesses are gonna take a long while. batch individual transfers and lay big files out contiguously on disk.

*2) Think about **caching**.*

- Hit time * hit rate + miss time * miss rate remember
- LRU policy - good for popular data, bad for sequential access.

*3) What about **concurrency**?*

- different orders of execution can lead to different performances.
- tradeoff b/w performance and fairness

*4) Multiple disks? how to parallel **divide**? well it depends on the bottleneck (see notes)*

REC6: The MapReduce tldr:

- specify a map function and a reduce function to parallelize computation.
- map - input key/value pair and produces a set of intermediate key/value pairs
- reduce - takes intermediate key I and all sets of values for that key then merges.
- master determines which workers take which task
- accounts for worker failure by tracking pings back to master, and if anything doesn’t report back, then completed tasks are re-executed by others
- straggler tasks - backup tasks

LEC8: Networking

- represent networks as graphs - endpoints on outskirts, switches in the middle (nodes). edges = direct connection
- Questions: *how do we address (location info)? how do we name? how do manage routing (the min cost route) and transport (sharing the network efficiently)?*
- History of the Internet
    - Sputnik - creation of DARPA/ARPA. Started off pretty small in the 70s
    - 1978 - making the Internet flexible - encourages a layered model
        - **Application - main program <-> called procedure (application protocol)**
        - Session/Presentation
        - **Transport** - reliable delivery -> TCP in 1983 implements reliable delivery
            - client stub calls some network procedure (send_message)
        - **Network** - IP (addressing routing)
            - forwarding data thru intermediate points to move it to the place it is wanted
        - **Link** - point to point links
            - moving data directly from one point to another
                - one package switch to another (literally sending that packet)
        - physical (physical medium)
    - Growth -> change in the 1980s
        - scalability. DNS. scalable routing
    - Growth -> Problems
        - Congestion collapse - TCP adds a congestion control mechanism.
    - 90's - policy routing because commercialization of Internet
    - Addressing - assign address in chunks of different sizes
- Problems we deal with today
    - DDoS, security issues - Internet was not designed with accountability or security.
    - address space depletion

REC7: The Ethernet TLDR

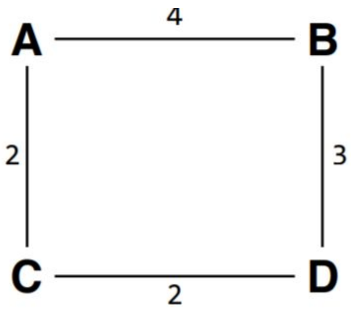
- goals - more scalable, and more reliable
- packet collision is a big deal - ethernet just retransmits at random intervals after
- packet addressing (dest, source, data, checksum)
- reducing probability of packet loss
    - carrier detection
        - nothing transmits while hearing carrier
    - interference detection - knowing your packet got interfered with

LEC9: Routing - Link-state, Distance vector

- Big questions - how do we route scalably, transport data scalably and adapt new techs?
- **General Routing Protocols**
    - *<u>Goal</u>: for every node, after the routing protocol is run, the node's routing table should contain a minimum cost route to every other reachable node.*
    - note that the <u>*route*</u> here means *the first motion on a path*. routing table can also have min path lengths on here. The path lengths are the sum of **link costs.**
    - **Routing tables** can help switches decide where to send packets!

A's routing table

```
routing_table[A] = self ; 0
routing_table[B] = A->B ; 4
routing_table[C] = A->C ; 2
routing_table[D] = A->C ; 4
```



- **Distributed Routing Protocols**
    - <u>3 step process - happens periodically to detect failures, changes.</u>
        - Nodes learn about their neighbors via the HELLO protocol
        - nodes learn about other reachable nodes via advertisements
        - nodes determine the minimum-cost routes
- **Link-state routing - disseminate topology information, and then run an alg like Dijkstras. good for small networks bc not good at scaling**
    - *Advertisement:* Each node advertises its list of neighbors and its link costs
        - this leads to **flooding**, where each node knows about every other node.
    - *Pros:* makes link-state very robust to failure. Nodes won't miss an advertisement.
    - *Cons:* Overhead on this is crazy af, 2NL ads (nodes, links)
- **Distance-vector routing - hard to reason about mistakes tho**
    - instead of full shortest paths, just keep the first hop
    - *Advertisement:* Each node's advertisement is a list of the nodes it knows about and the current costs to those nodes.
    - When updating, each ad will represent a node's cost to some destination.
        - If the source node is already using the node to get to destination, update the cost.
        - otherwise, see if the new node can provide a better path.
    - *Pros:* Overhead is much better - 2L advertisements
        - good for small networks
    - *Cons:* Counting to infinity

        - if the order of advertisements sending is weird, then nodes can incorrectly think there's a route when there isn't one if errors propagate. Then this can last for a while, since the alternate cost is infinity

REC8: Internet TLDR

- goals: continue Internet despite loss of networks
    - replication? LOLLL
    - keep info @endpts -> if the endpoint goes down, then its ok for it to lose its info [fate-sharing]. so gateways are stateless
- support a ton of services/networks
    - TCP and IP
        - at first they thought TCP could do everything but lol TCP sometimes is too cautious around failure
        - so IP is the general datagram model (UDP)
    - Internet also only makes a minimum number of assumptions (like the network will transport a packet or datagram of some size)
- to be distributed

Networks 101

- **point to point link -** wired things together, direct connection
- **switches -** devices that sit in the middle of the network and help move data from one machine to another
    - **multiplex** - multiple ways to send more than one stream of data thru a switch
    - **packet switching** has **headers**
    - **circuit switching -** relevant state is kept on the **switches**
- **addressing scheme is important**
- **routing protocol** - switch has a routing protocol, then it knows which switch to send the packet to next
- **routing protocols and addressing schemes don't need to worry about what's running underneath**
    - **"layering"**
- **switches contain queues. Full queues result in loss.**
- **Reliable transport** protocols try to deliver data more reliably
- **application layer -** like apps like Spotify or Skype
- we can think about network speed, bandwidth or throughput or latency.
- **multicast:** the ability to send a piece of data to multiple endpoints at once, not just one
    - **broadcast** if a network can do that
- **prioritization?** can a network allow for different treatment of different types of data?
- **policies?** rules for the network like prioritizing certain types of data?

LEC10: BGP Routing

- so we looked at distance vector and link state. but both are only really good for small networks.
- Question: How do we route scalably, while dealing with issues of policy and economy? **BGP!!!!**
- How do we deal with scale?
    - Routing hierarchy - divide the Internet into autonomous systems (**ASes**)
        - one protocol to route within, and one protocol to route outside (BGP)
    - path vector routing
        - like DV, but include the full path in the routing ad (overhead increases, but is still better than LS)
    - topological addressing
        - assign addresses in ASes contiguously, to make advertisements between ASes smaller
- **policy routing:** switches make routing decisions based on policies set by a human.
    - this is usually some financial gain
        - customer/provider - customer pays provider for transit
        - peers allows free mutual access to each other's customers
    - this is enforced by selective advertisements. maybe AS1 won't tell AS2 about a path.
    - BGP export policies -
        - providers export customer's routes to everyone ($$)
        - customer exports its providers routes to its customers
        - AS only exports customer routes to peers

REC9: The RON tldr

- IP/BGP tries to route something, but then RON is like lol too slow
- RON always tries to determine better routes via its overlayed nodes
    - so in a way, its just jumping over advertised policies from BGP. it takes advantages of links BGP won't advertise
    - takes advantage of path redundancy that BGP doesn't advertise
- works bc it moves fault detection and recovery to higher layer that is capable of faster response. it's also way more tied to the application using it (ie not scalable tho) and prioritizing latency or maybe throughput or low loss etc
- if theres a path b/w A and B, then theres a "virtual link" with some cost, that is basically the min cost of the underlying Internet links under it.

LEC11: Transport

- **Question:** How do we transport scalably now, once we've figured out how to route scalably?
- **TCP:** provide reliable transport, prevent congestion.
  - *Goal:* every app gets a complete, in order byte stream from sender. only one copy of every packet
- Idea: every data packet (1,2,3,4). Server sends some W outstanding packets at a time, and when a receiver gets a packet, it sends a cumulative ACK back. If sender doesn't receive an ACK back, it times out and resends the packet.
  - sender can infer from the ACK's received which packet was lost.
- Idea: **congestion control:** controlling the source rate to achieve high performance
  - Concept: minimize drops, delay and maximize utilization.
    - sharing of bandwidth- but how do senders know how many other senders there are?
  - **Signalling congestion - packet drops**
    - W = W+1 if no loss, = W/2 if there is loss (AIMD)
    - a conservative approach
  - some issues is how do we measure fairness?

REC10: DCTCP TLDR

- TCP causes a lot of timeouts, so its not really fit to data centers, which are latency sensitive
- TCP also only responds when packets are dropped, but by then, you're already screwed
- DCTCP - active queue management (sets ECN if queue is getting large)
- solves the problem of **incast (a ton of tiny flows into something at the same time)** by making queues small
- solves **the problem of long flows** taking up all the bandwidth by minimizing impact of long flows bc short queues for the completion of small flows

LEC12: Queue Management Schemes

- **problem:** TCP reacts to *drops,* and packets aren't dropped until queues are full T.T
- schemes
  - droptail - drop packets only when queue is full
    - probs is high delays and synchronized flows
  - RED - drop packets BEFORE queue is full
    - increasing drop prob as queue grows (but not when its full)
    - still drops packets though sometimes
  - ECN - mark packets and set a bit in the header to marked ACKs to let them know its getting there.

        - smaller delays, less oscillation
        - but its more complex, and more params
- traffic differentiation schemes
    - put different types of traffic in diff queues
    - priority queueing
- what if allocate bandwidth to different types of traffic?
    - round-robin - every round you take a certain number of bytes from each stream
    - weighted round-robin
    - deficit round robin

REC11: The end to end argument tldr

- you gotta have some features on the application side, not low level
- implementing low level introduces a lot of complexity and not enough generalizability and poor performance

P2P Networks + CDNs

- **P2P** - filesharing technology (more scalable than CDNs, which are more scalable than client-server[HTTP, FTP])
    - client downloads part of file from server, then uploads it to others
    - how to incentivize peers to upload
        - users can't DL from a user unless they're also UL'ing to them
        - protocol: round n -> some peers upload blocks to X. In round n+1, Peer X will send blocks to the peers that uploaded the most in round n.
    - how does this happen
        - torrent files contain info on tracker URL
        - peer contacts a tracker, which responds w IPs of other peers, and then the peer connects to other peers and transfers it to its neighbors, which then can connect to the remaining peers
        - seeders already have the entire file and are just sticking around for kicks
    - *problem*: tracker is a central point of failure? nowadays its just distributed hash tables
- **Skype -** VoIP
    - Consider A behind a NAT -
    - A -> N1 -> N2 -> S
    - but S's IP is private. Skype provides a directory, so we can get N2 IP

    - But when N2 gets packets from A, how does it know what do with them?
        - Employ a **supernode P,** that routes A and S through it and has a lot of state.

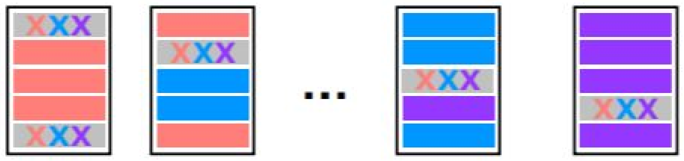
CDN tldr:

- problems - peering point middle man congestion, inefficient routing (BGP is bad), unreliable outages, TCP is also bad
- reliability
    - redundancy in edge servers, which communicates with end users, and theres a ton of them
- scalability
    - reflectors
    - tiered load balancing - caching in more edgey servers
- autonomy from humans
    - autonomous monitoring, recovery, assumptions of failure
- performance
    - optimizing comms
    - varies the protocol on context
    - packet loss reduction

LEC15: Reliability

- how to be a fault-tolerant:
    - <u>identify </u>all possible faults
    - <u>detect </u>and contain the faults
    - <u>handle </u>the fault
- questions to ask: how do we quantify faults? how bad is it?
    - **MTTF** = mean time to failure
    - **MTTR** = mean time to repair
    - availability = MTTF/(MTTF+MTTR)
    - today's version is how to deal with replication within a single machine
- ***How to deal with disk failures? bc if you mess up a disk, you suffer***
- ***RAID-1 try using two mirrored disks***
    - how to know something did the messup
        - on write, we write to both disks, and on read we read from either
        - so recovery will just be writing something from one disk to the other
    - eh: requires more disks (2N) T.T
    - yay: requires from single-disk failure
- ***RAID-4 try using a dedicated parity disk***

    - sector i of the parity disk is the xor of sector i from all data disks
        - so like if some disk A fails, then you can reconstruct disk A from knowing what disk B is and what P is (coming from the XOR)
    - good things?
        - we can recover from single-disk failure and it requires N+1 disks, not 2N
        - performance benefits
    - bad things?
        - all writes hit the parity desk - we update the parity disk all the time
- ***so just spread out all the parity between different disks***



-
- ***beautiful- we call this RAID-5. wonderful***

REC14: GFS tldr

- prioritizes fault tolerance and scalability
- assumptions
    - component failures are the norm
    - really big files
    - append only, not overwrite
    - reads are large streaming or small random
    - high bandwidth>low latency
- architecture
    - master and multiple chunkservers, accessed by multiple clients
    - files are divided into chunks and have chunk handles. chunk size is large.
    - master has all file system metadata. heartbeat messages
    - no one caches
- plan
    - client asks master who to contact, master tells them, client then caches this info for a bit and interacts directly w chunkservers
    - master stores file and chunk namespaces, mapping from files to chunks, and locations of chunk's replicas. first two are also kept on a log for consistency. the latter is polled at startup and is monitored via HeartBeats
    - operation log - critical, ergo replicated on multiple remote machines
- consistency, minimization of master role

    - master grants chunk lease to a replica (*primary)*
    - primary picks an order for the mutations to the chunk
    - workflow:
        1. client asks master for primary
        2. master tells client primary and locations of other replicas
        3. client pushes data to all the replicas. once receipt, client sends write request to primary.
        4. primary assigns serial numbers to mutations. then applies mutation to local state. then forwards to all replicas.
        5. secondaries ACK. primary replies to client
    - namespace locking, not file locking
- record append - appends are **atomic**
- snapshot
- MASTER
    - namespace management and locking
    - replica placement - maximizes data reliability and avail and network bandwidth util
    - garbage collection
    - stale replica detection
- reliability
    - chunk replication
    - master replication
    - data integrity - checksums on individual chunk servers

LEC16: What happens when replication fails us? you really can't replicate *everything.*

- question - how can we build reliable systems from unreliable components?
- **atomicity**: actions that either happen completely or not at all
    - idea: what if we use shadow copies? write to a tmp_file, and then rename the tmp_file as the actual_file? well what if you crash while renaming? then you're SCREWED
    - *key: make rename atomic. but that's messy*
    - interesting point - rename here is the commit point
    - what if you just **recovered from failure?**
- guaranteeing **isolation**
    - when multiple transactions run concurrently they appear as if they were run in series
    - *what if we used locks? but that scheme is not efficient*
- **transactions: defined by a begin and an end**

LEC17: Atomicity - building reliable systems from unreliable components

today on, making atomicity better than shadow copies

- the block is a *transaction,* which provides atomicity and isolation, while not hindering performance
- **atomicity -> shadow copies? its simple but its poor performance. what about logs? better performance, but slightly complex**
- shadow copies - like using local variable copies and then looking for errors instead of modifying in-place
- logs
    - **transaction syntax - begin -> write-> read-> abort/ commit!**
    - **idea of a transaction: keep a log of whether each action commits or updates or aborts**
    - **performance -** writes are good, but reads are *bad* bc you have to go through the whole log!
- idea - also use disk storage!
    - so now updates go to log (log) and storage (install), and reads go to storage
    - recovery - scan logs backward to determine what aborts were made
    - write-ahead logging - log then install. otherwise you cant recover between the two writes
    - **performance -** writes are ok (writing twice now), reads are good, but now recovery is bad!
- improving the log + storage idea
    - improving writing - write to a cache first, then flush out
        - potential problems include unflushed changes, but we can just redo thing in the log
    - improving recovery - truncate log
        - maintain checkpoint record in log to see when the cache was flushed

REC15: zfs tldr

- solves problems of partitions running out of space and writing now dominating overall performance
- automating partitioning the disk
- file systems should be decoupled from physical storage - multiple FS's should share one pool of storage

- keep data on disk self-consistent at all times - transition from one consistent on-disk state to another without any time when the system could crash
- checksums used for verification
- vdevs used to combine disks into storage pool
- copy-on-write: on-disk consistency, where old data is kept until the uberblock is overwritten (commit point)

LEC18: ISOLATION

now that we've abstracted to transactions and logs for atomicity, how do we enforce isolation?

- well why dont we consider single global locking? well you're screwed performance wise
- serializability: transactions "Appear" to run in sequence
- **final-state serializability:** final-state serializable if its final written state is equiv to some serial schedule
  - if for some possible sequential schedule, the order of operations leads to the same answer, then the schedule is final-state serializable
- **conflict serializable -**
  - two ops conflict: if two operations operate on the same object, and at least one of them is a write
  - a schedule is CS if order of all of its conflicts = order of conflicts in some sequential schedule
- **conflict graph**
  - edge from T_i to T_j iff T_i and T_j have a conflict between them (i,j being transaction #s)
  - conflict serializable iff exists an acyclic conflict graph
- **how to generate schedules that are CS? do we just generate all of them????? answer:** ***two-phase locking***
- 2PL:
  - each shared var has a lock, before any op, the transaction must acquire the lock, and after a transaction releases a lock, it can't acquire any new ones.
  - 2PL generates a CS schedule
  - dEadLOck?????
    - abort one of the transactions yolo
  - performance improvement: separate reader/writer locks

conflict graph - topo sort the DAG to find an ordering for the schedule

REC17: the database paper tldr

- **ACID - atomicity, consistency, isolation, durability**
- Algs/defs

- WAL - a protocol that says the commit is when you write something to nonvolatile storage
- 2PL - once you finish release a lock you may not acquire any more locks
  - guarantees a DAG
  - lock manager. also handles deadlock (avoidance or detection)
- **serializability, specifically conflict serializability**

- tradeoffs
  - phys vs logical logging
  - performance vs limited concurrency
  - optimistic vs pessimistic control
- stealing/forcing
  - **force [make sure after a commit, everything is in the hard drive, so no need to redo]** and **no steal [don't put anything on the hard drive before you've committed]**
  - **STEALS** imply you have to do **undos**
    - bc here you have values from uncommitted txns in nonvolatile storage
  - **NO FORCES** imply you have to do **redos**
    - bc here you have some updates that are not reflected on non-volatile storage after you commit
  - well why dont you do force no steal?
    - these imply more disk overhead
- physical vs logical logging = physiological logging
  - don't forget a commit is a nonzero amount of time!!!
  - common - steal no force, which is a weaker def
- locks
  - read/write, or long/short duration
  - levels
    - READ UNCOMMITTED - allows txns to read data that has been written by other txns that have not been committed
    - READ COMMITTED - txns only see updates from txns that have been committed
    - REPEATABLE READ - indiv data item reads above are repeatable (long duration)
    - SERIALIZABLE - protects against all the above, well-formed wrt reads on predicates, long duration lock
  - hierarchical locks - locks on whole pages so you dont have to grab a lock for every data item
    - txns can state intention

        - lock escalation - auto adjust granularity at which txns obtain locks based on behavior
- optimistic vs pessimistic (locking) concurrency control

LEC19: Transactions across Distributed Machines

client <-> coordinator <-> servers

- goal: develop a protocol that can provide multi-site atomicity in the face of all sorts of failures. today, instead of one machine, do multiple machines!
    - failures like message loss, reordering or worker failure
- how does logging work in this scheme?
    - once coordinator hears prepare ACK from all servers, it COMMITs
    - once it heard commit ACKs from all server, it DONEs
    - we really cannot abort after committing
- **two-phase commit -** nodes agree that they're ready to commit before committing
    - **what if we lose the prepare?** timeout, the send prepare request again
    - **what if we lose the prepare ACK?** timeout resend (thnx to seq #s, the server won't re-process it)
    - **what if workers fail while preparing?** everything aborts
    - **what if we lose the commit message?** server sends back tx status. and then timeout resend commit message
    - **what if we lose the commit ACK?** coordinator timeout resend
    - **what if worker failure during commit?** well we can't abort. so it has to recover into a prepared state (LOGGING). then send back the tx? and commit
    - **what if coordinator fails during prepare?** after recovery, everything aborts
    - **what if coordinator fails during commit?** well we can't abort. so once coordinator recovers, we recommit
- well ok, but when a worker fails, then isn't some of our data unavailable?
    - well we could use replication. but how will we keep copies consistent? how to achieve single-copy consistency?

REC18: consistency guarantees tldr

- eventual consistency - any combo of values in the past is possible (like R1, R3, R9)
- strong consistency - always get the last value
- bounded staleness - get some value from last t seconds
- monotonic reads - any value in past, but once you read some R1, R2 is either the same or after R1
- read my writes - be able to see most recent stuff from ur own writes
- consistent prefix - be able to see all values [0, R_n]

LEC20: Distributed Transactions - Availability and Replicated State Machines

Those rules

1. Primary must wait for backup to accept each request
2. Backup must reject direct coordinator requests
3. Primary must reject forwarded requests
4. Primary in view i must've been primary or backup in view i-1

- replicated servers can become inconsistent
- what if we had coordinators who communicated with primary servers, who communicate with backup servers? and then if the primary fails, they go to the backup
- well if you have multiple coordinators, you will have problems. the network partition may force two different things to be the primary for each coordinator, and then since theyre using different primaries, they aren't consistent anymore!!1!!111
- **use a view server**
    - determine which replica is the primary
    - tells coordinator which one is primary
    - primary/backup will ping the VS so to signal failures
    - a lack of pings indicates a server is down
- **primary failure due to partition**
    - well now the VS is going to think some S1 is dead because of a partition. S1 is still primary to S2. But now VS makes S2 primary. [slide 30]
    - ok but hold on now we have problems
    - What happens after S2 knows its the primary, but S1 also think its the primary? [slide 34]
- **VS failure -** we cry and give up

REC19: Raft tldr

- prioritizes understandability
- strong leader that manages replicated log
- leader election via random timers
- membership changes

- state - leader follower candidate
- Appending process:
    - leader appends command to its log as a new entry
    - issues AppendEntries to all other servers
    - once ACK'd, leader applies entry to own machine and signals client
    - otherwise, retry indefinitely
- stuff is committed once entry is stored on majority of servers
- reelections happen when followers don't hear a heartbeat from leader for a while
- if follower/candidates crash, its ok because the leader just keeps resending the RPC until it comes back online.

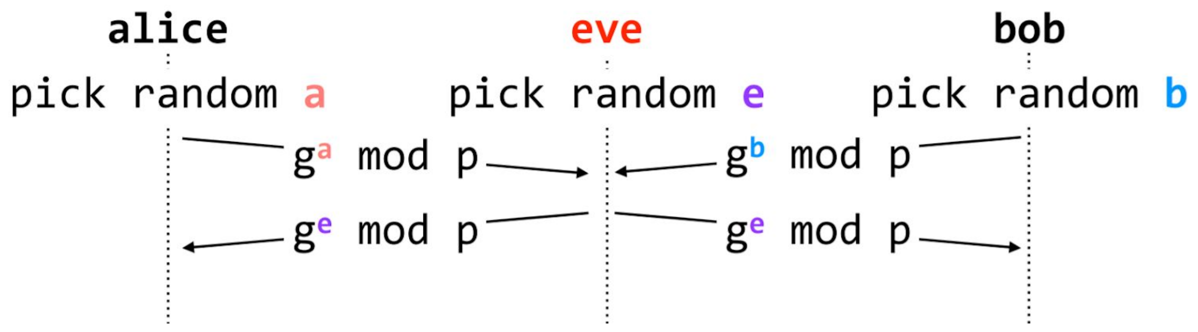
LEC22: Authentication - how do we verify that the user is who they say they are?

- use passwords
- how to implement them?
    - Threat model: attacker has some access to server on which pw info is stored
    - idea 1: store plaintext! bad af! don't do! this!
    - idea 2: store hashes of passwords on the server
        - p good.... but adversaries can compare hashes of popular passwords.
        - like they can figure out hash of “123456” and then figure out who has that pw
        - this is called a rainbow table
    - idea 3: salt the hashes
        - store username, a salt (Random #), H(pw + random #)
        - harder to build a rainbow table
- session cookies to verify users?
    - client sends user/pw. server sends back a cookie
    - cookie = {username, expiration, H(serverkey | username | expiration)}
- phishing attacks - adversary tricks users into visiting a legit looking site and then gets their user/pw
    - solution 1: challenge-response protocol
        - server gives client a random value r
        - client computes H(r+pw) and sends back to server
        - server compares with expected pw
    - note on solution 1, if server stores salted hashes
        - client sends H(r | H(s | p))
- bootstrapping/resetting passwords
- consider instead - pw managers, two-step verification, biometrics

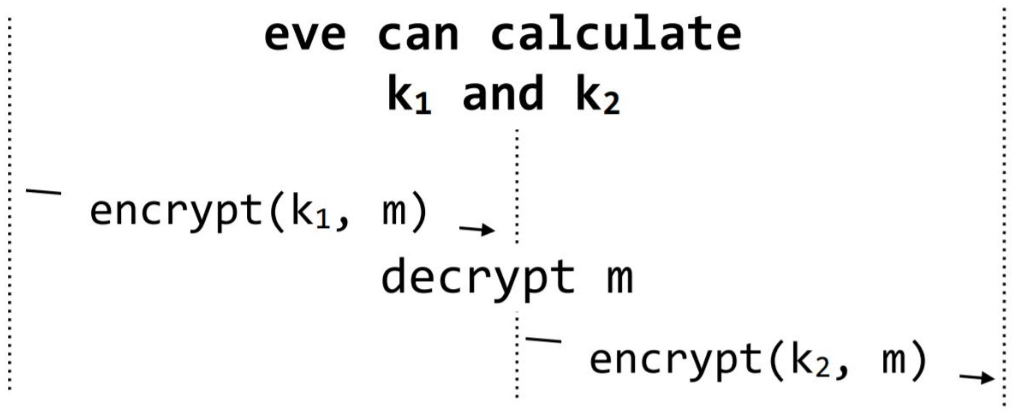
REC20: cryptosystems fail tldr

- why are we so silent on crypto problems
- ATM fraud takes place when
    - insider attacks (people are bad people)
    - outsider attacks (people can steal your card info)
        - replay attacks
        - programming errors
        - false ATMs built literally
    - poor PIN entropy, poor PIN distribution
    - poor PIN storage
- ATM encryption -> account # + PIN key = result of DES -> natural PIN + offset = customer PIN
    - use a security module to manage all bank keys and PINs ST programmers only see them in an encrypted form
- problems w/ encryption products
    - lots of people just dont ever bother to use actual security modules and they implement encryption in code, which is highkey bad
    - some people don't even set up master keys right
    - some people use bad encryption algs
- tldr the threat model was wrong - they thought PIN stealing would be the worst thing, but turns out people was the worst thing

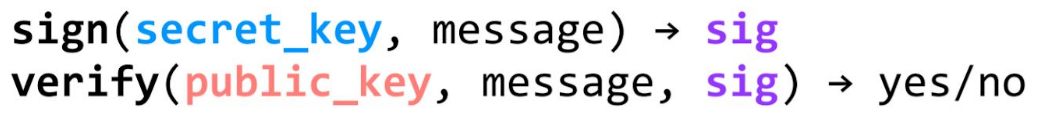
LEC24: network security



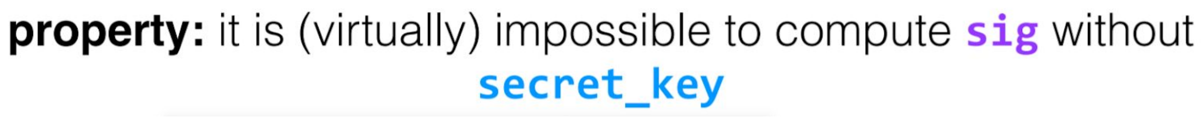
eve here is trying to listen to alice’s and bob’s stuff, so she send back her encrypted key (g^e mod p)



MITM attack,.... and they don't even know it happened! they don't know theyre not communicating directly
solving this ->



```
sign(secret_key, message) → sig
verify(public_key, message, sig) → yes/no
```



property: it is (virtually) impossible to compute sig without secret_key

you **gotta** know secret key to get sig. so this prevents MITM, since eve can't generate sig.

distributing public keys:
signed messages sign({Bob, PK_bob},{secret_key_CA}) - certificate authorities

TLS - client/server use public key crypto to exchange a secret, and then they use the secret to generate keys for symmetric crypto

topics: asymmetric vs symmetric encryption

REC21: dnssec tldr

- provides data integrity and origin authentication via signatures
- each zone has a PK, SK pair
  - more on architecture, each zone needs to provide a primary and secondary NS to enhance resiliency.
- each parent zone signs child PKs with parent SK. chain of trust
- DNS issues
  - MITM

  - packet sniffing (capturing queries can generate wrong answers a lot faster than actual NS responses)
  - transaction ID guesses
  - cache poisoning using RDATA portion in a DNS name
  - DDoS attacks
  - zone transfer by attacker
- DNSSEC is backward compatible
-

LEC24: Network Security and DDoS attacks

- ddos attacks - congest the service, mounted from different machines
- botnets can be huge collection of machines that are compromised
- how do we know which IP addresses are part of the botnet to block?
  - NIDS (Network Intrusion Detection Systems)
    - signature-based detection - good for preventing known attacks, bad because can’t detect new attacks
    - anomaly-based detection - match traffic against a model of normal traffic
      - good because we can detect new attacks, but how can we model traffic well?
  - as an attacker, how to evade NIDS
    - can confuse state on NIDS
    - mount an attack on the detection mechanism
  - we can also look like legit traffic
    - HTTP flooding
    - TCP SYN floods - exhaust state on server
    - optimistic ACKs - attacker ACKs packets it hasn’t received yet
    - DNS reflection/amplification
      - spoof requests to cause large DNS responses to be sent to victim’s machine
- attacks on routers
  - overload routing tables/CPU
  - hijack prefixes ST attacker gets an AS to announce that it originates a prefix that it doesn’t actually own. or like advertising a shorter route
    - one could secure BGP like DNSSEC, but thats a big performance hit

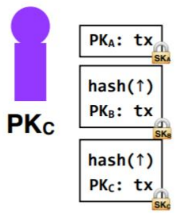
REC22: botnets

- torpig is a botnet (network of malware-infected machines controlled by adversaries) that harvests a ton of data
- studying botnets can consist of passive analysis, active analysis, or hijacking
- how it infects
  - victims infected via drive-by-download

    - Mebroot is loaded from the MBR upon reboot and contacts Mebroot C&C server to obtain malicious modules
    - server can send a config file to the bot
    - Torpig uses phishing attacks
- domain flux
    - DGA computes a list of domains, and the bot tries to contact each one until one works
    - DGA seeded w/ date and numerical param
- they registered a domain that the bot would resolve

LEC25: Blockchain and Bitcoin

- Distributed public logs/ledger - the blockchain
    - users append transactions to log. everyone has a copy of the log. decentralized.
    - **challenges:** how do we ensure anonymity? How do we maintain integrity of transactions? How do we prevent adversaries from duplicating/removing/reordering transactions? How do we maintain consensus?
- ensuring anonymity: users identified only by public keys
- integrity: users sign their transactions
- preventing reordering: include a hash of previous transaction as part of signed



  data
- What if we have bad people spewing fake stuff all over the place?
- **Consensus question:**
    - idea 1: broadcast transactions, wait for majority of network to confirm them. nope this is bad bc you can just make a ton of identities
    - idea 2: user receives two or more copies of log and they conflict, then keep the one thats the longest and forward it along. this still got messed up with above attack
    - one way to fix this is to make it expensive to validate a block.
    - mechanism to validate blocks (yes this mines bitcoin):
        - check in the log that transaction is valid.
        - once it is, Find nonce ST H(t|nonce) < target, this takes like 10min.

        - publish and broadcast
        - and yes each block contains multiple transactions

REC23: do you need a blockchain?

- permissionless blockchain
- permissioned blockchain - only some limited set or readers and writers
- props
    - public verifiability (allowing anyone to verify correctness of system)
    - transparency
    - privacy
    - integrity
    - redundancy
    - trust anchor - highest authority
    - privacy vs transparency: can a distributed ledger provide public verifiability of the overall state without leaking info about the state of each individual participant?

LEC26: Tor - how to solve anonymity

- Tor - network for users to remain anonymous
- Bitcoin - users can possible remain anonymous
- Tor mechanism
    - Alice -- [to:proxy|From:Alice|Data] --> proxy server --> [to:server|from proxy|data] --> server
    - responses also go thru proxy server
    - problems: proxy knows that alice is communicating with server
    - so have multiple proxies!!!!!! each node has previous and next hop
    - problem: what if adversary tracks packet data and sees data between A and P1 and data between P3 and S are the same? and conclude it's A
- Tor - network of proxies + encryption
    - each proxy gets a keypair, and each proxy strips off a layer of encryption

```
       A -- [to:P1|from:A|PK_P1(circuit:K|PK_P2(circuit:K|
PK_P3(circuit:K|XXX)))] -->
       P1 -- [to:P2|from:P1|PK_P2(circuit:K|PK_P3(circuit:K|XXX))] -->
       P2 -- [to:P3|from:P2|PK_P3(circuit:K|XXX)] ------------------->
       P3 -- [to:S|from:P3|XXX] ------------------------------------->
```

    - S
- possible attacks
    - traffic-correlation: taking into account the traffic in and out of the node, you can see packet sizes, timing and infer who is communicating with Serve

- performance is also kinda bad