16 Multi-Paxos, Raft & NewSQL Consistency across replicas

1. CAP theorem: 一个分布式系统，最多只能满足三个特性中的两个
    - Consistency: 所有的节点看到的数据是一样的
    - Availability: 每个请求都能得到一个响应
    - Partition tolerance: 系统在网络分区的情况下，仍然能够工作
2. 后面讲的就是 raft 了
    - paxos 是 bottom-up 的（先保证单值）
    - raft 是 top-down 的（直接解决多个 log 一致的问题、不用先解决单值一致的问题）
    - raft 和 paxos 是等价的协议
    - 一个值被 majority 了、也会在 raft 里被改
3. raft 的分解：
    - Leader election
        - Select one server as the leader
        - Detect crashes, choose new leader
    - Log replication (normal operation)
        - Leader accepts commands from clients, append to its log
        - Leader replicates its log to other servers (overwrites inconsistencies)
    - Safety
        - Keep logs consistent
        - Only servers with up-to-date logs can become the leader
4. raft：通过状态机 RSM+log
    - 三种状态：leader、follower 和 candidate
    - Leader: handles all client interactions, log replication
        - Invariant achieved: At most 1 viable leader at a time
    - Follower: passive (only responds to incoming RPCs)
    - Candidate: used to elect a new leader
    - 所有 log 只能由 leader 能做（限制），每个 term 里面只有一个 leader，leader 挂了会选一个新的
    - 为什么不能直接变成 leader 呢、因为会有 primary backup 的不一致问题
    - 只有 leader 可以发起 log 的 append，然后 follower 会把这个 log append 到自己的 log 里面
5. raft 通过 heartbeat 来决定是否需要 election（leader 挂了）
    - 如果收到了大多数 server 的投票、就变成 leader
    - 每个 server 只能投一票
    - 也是有一个 term 号、和 paxos 里的 proposal 号类似
    - 如果没人赢、就重试
6. election 需要 saftey 和 liveness

- liveness：有可能从 candidate 里面永远选不出：解决方法：用一个随机 timeout 的 heartbeat
- 每台机器只需要记住 currentTerm 和 votedFor（对于 election）

7. Log entry = index, term, command （log[]）
   - commit 就是把 log 输入到 statemachine 里
   - crash 的时候会造成 log 不一样的问题
   - 所以有两步走：先把这个 log 变成大概率是一样的、然后在这些大概率是一样的 log 的里面去做 consistency
8. 它保证的是一个 prefix 的 coherence
   - 就是如果 prefix 一样的话、才 appendEntries
   - 如果 prefix 不一样、就不会 appendEntries
   - 如果 prefix 不一样的话、会由 leader 发起进行一个覆盖的操作
   - 如果有很多不一致、所以就会递归覆盖、就是全部覆盖
   - 所以会导致 majority 写了、还是最后被覆盖了
9. 实现：
   - 保证这个 leader 是 truth
   - 如果不一样、就把其他的覆盖掉、变成 leader 的 log
10. 如何保证一个 log 被 commit 之后不被 overwrite 呢?
    - 可以避免一个比较少的 log server 变成 leader（选 leader 的时候加限制就行了，要有最新的 term 和 index 数，先按照 term 比、再按照 index 比）
    - 但是 ppt 上有一个例子、就是会覆盖掉 majority 的值，但是也有解决方法
    - 比如说、不能让 S5 成为 leader、就不会覆盖掉这个 2 了
    - 一旦这个 log 被 commit 了、所以就要保证这个 log 在 future leader 里面
    - 所以在选举 leader 的时候、加一个限制就可以了(lastTermV > lastTermC) ||(lastTermV == lastTermC) && (lastIndexV > lastIndexC)

17 Introduction to Network

1. OSI, TCP/IP & Protocol Stack
   - OSI 有 7 层
   - TCP/IP 有 4 层（最常用）
   - CSE 只 care 3 层
2. Layers in Network
   1. Application
   - Can be thought of as a fourth layer
   - Not part of the network
   2. End-to-end layer
   - Everything else required to provide a comfortable application interface （解决数据传输出问题的问题）
   3. Network layer

- Forwarding data through intermediate points to the place it is wanted (找节点的邻居，怎么找到一条线连接到目标节点)

4. Link layer
   - Moving data directly from one point to another （两个节点直接相连）
3. UDP 更关注的是速度，不保证发出的包一定能收到
4. 因为这个包的数据是往前增长的、所以会用 socket buffer 预留前面的空间（就是先分配包的空间、然后再填数据）
5. transport layer
   - 为了区分然后建立多个连接、所以会有 port（共 65535 个）
   - 建立的连接数是有限的
   - 握手的时候的 seq number 是根据直到当前发的数据决定的（就是这个 seq number 是随机的）、所以可以防止第三方进入
6. network layer：
   - 解决怎么找到一条路径的层的问题
7. The Link Layer

   The bottom-most layer of the three layers
   Purpose: moving data directly from one physical location to another

   1. Physical transmission
   2. Multiplexing the link（一条线怎么传多个数据）
   3. Framing bits & bit sequences
   4. Detecting transmission errors
   5. Providing a useful interface to the up layer
8. 传输数据的方法：
   1. （同步的方法）用 shared clock，时钟上升沿的时候就去数据线上读数据（但是需要采样很精确，容错空间小、数据的周期也需要和时钟一样）（只在 cpu 内部这么传）（两条线、时钟线和数据线，但是两条线需要高度的协同）
   2. 不用 shared clock、可以用异步传输（ppt 过程，一开始 ready 和 ack 是 0，data 是 0 或 1，如果要传数据了、就把 ready 设成 1，然后 b 看到 ready 了、就读 data、然后把 ack 设成 1，a 看到 ack 是 1 了、就设 ready 是 0，b 看到 ready 是 0 了、就设 ack 是 0，然后结束）（好处：不需要时钟，不需要对表、而且采样不会出错、因为 ready 了才读数据）（缺点：是 2Δt （每次数据传递需要 Δt ）才能传一个 bit）（解决方法：一次传 N 个 bit，parallel transmission，比如 data 不是 1 个 bit、data 传 64 个 bit）
   3. serial transmission（usb 使用）（有时候这个波形会变形，所以会需要使用压控震荡器，能恢复出信号对应的时钟周期）（问题在于有时候这个信号没有变化、所以需要编码）
   4. 并行的接口有问题：数据传过去、每一处都有一个 bottleneck、会受限制，而且会有线的电源干扰、所以这个就是 usb 的使用（usb 就是串行的快速变化的一根线）
9. VCO：压控传感器、从数据信号中恢复时钟，因为有时候这个数据会失真（但是这个需要这个变化的波形、如果有大量的 0 或者大量的 1 的时候、就会有问题、所以需要下面的曼彻斯特编码）
10. 曼彻斯特编码：
    - 本质是把变化嵌入到了每一个 bit 里面
    - 0 -> 01
    - 1 -> 10

- 还有一个 8B10B 编码（8 bit -> 5 + 3 bit）（10 bit -> 6 + 4 bit）（就是把 8b 的编码变成 10b 的编码）（111100，1110）（0001）（好处就是增加变化）（数据率高一点）
- 缺点：数据率只有 50%

11. 如何共享一个连接
    - Isochronous（同步的复用，第一种方法，现在用得很少）：对一条数据线分时复用（好处就是这个 8bit frame 是预留的，能保证语音的质量）（坏处是这根线上只能有 703 个同时的 conversations，因为带宽有限，超过 703 就会拒绝接入）
    - 相当于是 5624 / 8 = 703
    - 所以会有一个 multipexing/demultiplexing 的过程，可以把包放在一个 buffer queue 里面（单纯加内存不能解决问题、因为会导致等待的时间变长、然后就会 timeout、然后会重新发包、然后会导致时延变长）（所以丢包是一个好的选择）（因为发包的速度远大于收包的速度）
    - 有优先级的问题，比如两个包同时到、收哪个包
    - 还有一个 Frame and Packet: Asynchronous Link 的方法，用发包的方式进行数据传输（变长的数据包，使用包交换的方式）
12. 包头和包尾的作用：识别什么时候开始和结束一个 frame
    - 怎么通过 01 判断有一个新的 frame 来了，所以通过编码的方式来解决这个问题
    - Simple method
    - Choose a pattern of bits, e.g., 7 one-bits in a row, as a frame-separator
    - Bit stuffing: if data contains 6 ones in a row, then add an extra bit (0) （需要对数据做额外处理、如果数据里面有 6 个 1，就加一个 0）
13. Error Detection
    - 如果传错了、就需要重传、有额外的代价。
    - 首先、用一个 checksum 来检测错误
    - 最好也能够纠错：所以需要冗余的思想
    - hamming distance：两个 bit 之间的距离（两个 bit 之间的差异的数量），可以通过 hamming distance + simple parity check 来监测错误
    - hamming 距离越大、合法值之间的变化就越不可能发生
    - 4bit -> 7bit 中为什么要在 1，2，4 位上，因为这样就能够推出 3，5，6，7 位的错误的位置（为了电路设计）（不过错了两位就没办法了）（就是做异或）
    - 然后通过计算 P1，P2，P4，然后查表、如果是哪些组合出错了、那么就是哪个位错了
    - 问题是，错了一位可以纠错、但是如果错了 2 位、可以检测、三位有可能能监测出来
    - 缺点是效率低、数据利用率低

18 Network Layer All about routing

0. IP: Best-effort Network
    1. Best-effort network（尽力而为）
       If it cannot dispatch, may discard a packet
    2. Guaranteed-delivery network
       Also called store-and-forward network, no discarding data（存储转发网络，不会丢包）
       Work with complete messages rather than packets
       Use disk for buffering to handle peaks

Tracks individual message to make sure none are lost
In real world
No absolute guarantee
Guaranteed-delivery: higher layer; best-effort: lower layer

1. 网络传输中的所有数据不是同样重要的、比如包头信息会更重要（所以包头后面会有 checksum）
2. The Network Layer
    - Network attachment points 网络接入的节点（AP）
    - Network address
    - Source & destination
3. 为什么 send 对应的不是 receive，而是 handle
    - 因为还可能会对这个包做转发
4. Managing the Forwarding Table: Routing
    - 每个门就是一个端口
    - 所以需要一个表去记录每个包要去哪
    - 也有一个截获流量的例子（这个表是会变的）
    - Static routing vs. adaptive routing 就是找到一条路线，而且要考虑到路线上面节点的状态
5. IP Route Table
    - 简单的是 network-interface pair
    - 但是实际的路由表很复杂
    - 落在不同的网络段、就从哪个网络口出去
    - 里面存的是下一跳的地址+端口?
6. Control-plane VS. Data-plane
    - 所以把这个网络层的任务分成了两个部分，一个是 control-plane，一个是 data-plane
    - Control-plane：如何构造这张表（和性能相关度不是特别大，要求 adaptive 就可以了）
    - Data-plane：如何通过这张表去转发数据（性能相关度很大）
7. 每台路由器上的路由表都是不一样的，会更多关注 local 的节点
    - 并不存在一张全局的表，因为没人能存的下，而且也没有必要存那么多
    - Allow each switch to know, for every node dst in the network, a route to dst
    - Allow each switch to know, for every node dst in the network, a minimum-cost route to dst
    - Build a routing table at each switch, such that routing_table[dst] contains a minimum-cost route to dst
    - 所以需要让这个全球的网络可以通过这个表的转发相连
8. Distributed Routing: 3 Steps in General
    - Nodes learn about their neighbors via the HELLO protocol（敲门）
    - Nodes learn about other reachable nodes via advertisements（把自己的邻居告诉别人）
    - Nodes determine the minimum-cost routes (of the routes they know about)（要做的、就是找到最短的路径）
9. Two Types of Routing Protocol

    - 第一种叫做 link-state routing 一个是通过 flooding 所有的 neighbor 状态（广告里面包含了 neighbor 和到 neighbor 的 cost）（然后通过 Dijkstra 算法找最短路径）（交互的内容少、但是发给所有人）
    - 第二种叫做 Distance-vector Routing 第二个是只告诉 Nodes advertise to neighbors with their cost to all known nodes（只给自己的 neighbor 发包、但是信息量大）
10. W 是还没有处理过的节点
    - 然后讲了一下 link-state routing 就是用了 Dijkstra 算法去找最短路径
    - 通过 dijkstra 不断更新这个表
    - flood 对 failure 的容忍还是比较高的（因为是全都发）（如果有一个节点挂了、通过 flood 的机制、所以还能到达其他节点就可以了）
    - 但是 flooding 的 overhead 很高、因为会充满这个邻居数据的包（n 平方）
11. Distance-vector Routing
    - 所以为了降低上面的 overhead，所以只给自己的 neighbor 发包
    - 每个节点收到自己 neighbor 的 advertisement
    - A's neighbors do not forward A's advertisements; they do send advertisements of their own to A（不给邻居发包含它自己的信息）
    - 相当于 bellman-ford 算法，所以只做一轮得到的结果不是最优解、所以要多次迭代后得到最优解
    - 但是对 failure 的容忍度不高、因为只是告诉邻居
    - limits 是 failure handling
12. 有一个 infinity 的问题、就是这个节点不可达的时候、会不停地给邻居发值、导致这个值变成无穷大
    - 通过 split horizon 的方法来解决这个问题，即这个节点不会把来自源的信息再发给源
    - 从一端收到的信息、不会再发回去
    - 就是发过的重复信息、不会再发、只发更改过的信息、所以 a 不会发没改过的 bc 信息给 b
13. Link-State Summary
    Pros:
    Fast convergence
    Cons:
    flooding is costly: 2 x #Node x #Line advertisements
    Only good for small networks
14. Distance Vector
    Pros:
    Low overhead: 2x #Line advertisements
    Cons:
    Convergence time is proportional to longest path
    The infinity problem
    Only good for small networks
15. 上面这个 Distance Vector 和 Link-State Summary 的方法都只适用于比较小的网络、因为会有很多的 overhead，怎么 scale 呢?
16. 3 Ways to Scale
    - Path-vector Routing（不记一个点、记一条路）

      Advertisements include the path, to better detect routing loops

- Hierarchy of Routing

  Route between regions, and then within a region

- Topological Addressing（让 hierarchy 反映出它自己的结构）

  Assign addresses in contiguous blocks to make advertisements smaller

17. Path Vector Exchange
    - 网络中的每个节点都维护一个 path vector
    - 也是通过 advertising 进行更新
18. 这里的例子是关于站在 G 节点的视角的
    - 通过 path vector 的方法、可以降低收敛的时间
    - 为什么好，因为有一个 path 在里面
    - 也提到了几个对 path vector 的问题的处理方法
19. How do we avoid permanent loops?
    - When a node updates its paths, it never accepts a path that has itself （就是这个更新路的时候、一条路里面没有自己就行）
    - What happens when a node hears multiple paths to the same destination?
      - It picks the better path（同样的路时、选更快的）
    - What happens if the graph changes?
      - Algorithm deals well with new links
      - To deal with links that go down, each router should discard any path that a neighbor stops advertising （把挂掉的点的路给删了、重新快速收敛）
20. 通过 region 的方法做 Hierarchical Address
    - 即相当于做 local 的 routing，然后再做 global 的 routing（层次结构）
    - 可以大大压缩路由表的大小
    - 但是缺点就是更加复杂了、而且需要把 region 和 ip（地址） 进行绑定了（因为需要通过 ip 快速定位到 region）
    - 也可以有很多层次的 region
21. hierarchy 的问题
    - Problems introduced by hierarchy: more complex
    - Binding address with location （因为相邻的地址做编码会方便）
    - Has to change address after changing location
    - Paths may no longer be the shortest possible
    - Algorithm has less detailed information
    - More about hierarchy
    - Can extend to more levels
    - Different places can have different levels
22. Routing Hierarchy
    - 跨 region 的路由需要用到 BGP （Border Gateway Protocol）
23. Topological Addressing

- 类似于掩码的逻辑 18.0.0.0/24（CIDR notation），从 18.0.0.0 到 18.0.0.255 都是从一个端口出去、这样就可以压缩路由表
- CIDR: Classless Inter Domain Routing
- 就可以用一条去表达很多条
- 为了 scale：所以要让路由表变小

24. 上面讲的都是 control-plane，下面讲的是 data-plane
25. data-plane 里面只需要考虑单机了
    - handle 可能还需要调用 send、所以不是 receive
    - net_packet.destination != MY_NETWORK_ADDRESS 的话、才需要 link_send
    - 就是如果是 localhost 的话、就不用 send 了
    - 如果收到一个不是自己的包、那自己可能就是路由器（就需要把包转发啊）
    - 一共有四条路径：收包的时候、可能是自己的或者不是自己的、发包的时候、也可能是自己的或者不是自己的，用一个 handler 判断需要往哪里走
26. forward 一个包的时候、需要做很多事情
    - 先查表
    - TTL：在网上能够通过多少个路由器转发、防止无限循环，如果 TTL 是 0 的话、就把这个包丢掉
    - (Time To Live) TTL：能在网络上通过多少个路由器进行转发
    - 更新包的 checksum（因为要减少 TTL）（有专门的硬件进行加速）
    - 然后再转发
27. Data-plane Case Study: Intel's DPDK
    - 是 bypass 了 kernel
    - RX 和 TX 分别代表了收包和发包
    - 直接访问网卡中的内存空间
    - 是通过轮询 hardware queue 的方式去做的、就没有 kernel 的介入（很快，没有中断）
28. RouteBricks
    - 用了很多个便宜的网卡机来代替了一个昂贵的路由器
    - 降低了成本
29. NAT (Network Address Translation)
    - Private network Public routers do not accept routes to network 10 (e.g., 10.8.8.8)
    - 因为网络的出口比较少、所以会不够用
    - 就是在发送的时候、把自己发出去的包的内网 ip + port 翻译成一个公网 ip （相当于是 router 的 ip）
    - 然后在接受的时候、再把这个公网 ip 翻译成内网 ip + port
    - 其实不太好、因为在网络层用到了不是网络层的 port 的信息、破坏了层和层之间的封装性（需要更改 payload 里的 port）
    - port 是 end to end layer、让你去复用一个网络
    - 就是消耗路由器的端口、然后 router 的端口是有限的、相当于是一个端口对一个端口
    - NAT 是有限个的，如果 10 个笔记本连一个路由器、最多每个分到十分之一个 NAT
    - NAT 表会成为瓶颈

- 对于 IP-set，不能经过 NAT、因为是 payload 里面的 port 是加密的，所以不能用 NAT
- ftp 也不能经过 NAT（其实也破坏了封装性的设计）（因为把 ip 写到了数据层 data 里）

30. NAT router: bridge the private networks
    - Router between private & public network
    - Send: modify source address to temp public address
    - Receive: modify back by looking mapping table
31. Limitations
    - Some end-to-end protocols place address in payloads
    - The translator may become the bottleneck
    - What if two private network merge?

19 End-to-end Layer Best-effort is not enough

1. 以太网 Ethernet
    - 就是需要监听所有的包、不能有冲突
    - 需要监测一个人发包的时候没有其他人在发包
    - 这个包不能太小、需要有 padding、不然会导致传输失败（最小 368）
2. 以太网发包的时候、拓扑结构有两大类
    - Hub：所有的包都会发给所有的人（广播）（意味着可以做监听）（A 10/100Mbps hub must share its bandwidth with each port）
    - Switch：会通过路径的调转，只会发给目标的人（不会广播）（Keeps a record of the MAC addresses of all the devices）
3. 以太网广播的时候，handle 函数里面就不需要转发了
4. Layer Mapping:
    - 例子里的上面这些 block identifier 都是 ip 地址、下面的数字 ethernet 是 mac 地址
    - mac 地址：48 bits
    - 在 ip 层发的时候、也会在包头包含 ip 的 source 和 destination 的地址
    - 到 ethernet 层、通过 mac 地址进行传输（相当于在包头填了一个 mac 地址）
    - 但是会有一个现象、就是 mac 地址是可以重复的，所以会有不同的设备有同样的 mac 地址
    - 自己的 mac 地址也可以修改，多个 mac 地址是可以一样的（只有 ip 需要唯一、而且 mac 可以改）（所以只要在同一个网段不重叠不重复就可以了）
    - 例子里面、如果要发包给 E、就要设置目标的 ip 地址为 E（E 的 ip 地址），然后设置目标的 mac 地址为 router K 的 mac、即 19，这样 K 才能收到这个包
    - 然后 router 从 4 号口发出去
5. layer mapping（arp cache）
    - 会在表里面存储 ip->mac 的映射
    - 如果发给网段之外的、就是 mac 就是路由器的 mac
    - 之前上面的 routing table 是通过 hello protocol 和 advertisement 来更新的
    - 这里使用了 ARP 协议（Address Resolution Protocol）来解决当一个新加入网络的时候、如何找到这个新加入的设备的 ip 地址和 mac 地址的映射，就是新加入节点能有这张表

    - 会通过广播的方式来找到这个映射（如果有人回应、就会更新这个表）（这里的表是存在了每个设备上。这个存的 station 为啥是 19？因为是路由器）（貌似每个机器都会存一个 cache）
    - 这个表就像 cache 一样、会不断更新
6. RARP：Reverse ARP
    - mac -> ip
    - 通过 mac 地址找 ip 地址
7. gateway 貌似就是 router 的意思，这里有一个例子
    - 在例子里面，会先填 target mac 为 gateway 的 mac
    - 然后根据 route table 修改接下来的包的 source 和 target 信息
    - 所以 link layer 通过修改 mac 地址来完成相邻节点的包的传输
    - source ip 也会改
    - 就是只用 mac 地址来完成两个点之间的传输
8. 会有一个 arp spoofing(投毒) 的攻击
    - 核心方法是 arp poisoning（本质上是对 arp 表的污染）
    - hacker 在没人问路的情况下广播很多消息，把正确的 ip-mac 映射关系给替换掉，然后这个包就被发到了 hacker 的机器上
    - 相当于 hacker 会变成一个中间人，可以看到所有的包（在不需要路由的场景下、hacker 把自己变成了一个路由器，就可以监听）
9. 如何防止 arp spoofing
    - 这个主要是因为协议问题、所以会比较难解决
    - 方法是：定期监听网络中的 arp 包的流量，然后发现有异常流量和数据的包的时候、就会发警告。但是没法根本性地解决。
    - 还有一个方法是：用静态的人为设定的 arp 表，但是这样的话、就会导致网络的可扩展性变差
10. 上面讲的都是网络层的问题，下面讲的是 end-to-end 层的问题
    - NAT 和 ARP 都是在网络层中会发生的变化
11. 网络里 如果网络不同会怎么办？考试?
12. end-to-end 层
    - 因为 network layer 没有保证很多东西、所以需要 end-to-end 层根据不同的应用场景进行设计（Delay - Order of arrival Certainty of arrival - Accuracy of contentRight place to deliver）
    - 没有 one-size-fits-all solution
    - TCP 保证了数据的完整性、但是 UDP 不保证数据的完整性
    - TCP 的 flow control 是可以当数据发得太快的时候、控制发送的速度
    - 这里介绍了 UDP、TCP 和 RTP
13. end-to-end 考虑了 7 种问题
    - Assurance of at-least-once delivery
    - Assurance of at-most-once delivery
    - Assurance of data integrity
    - Assurance of stream order & closing of connections
    - Assurance of jitter control（实验）

- Assurance of authenticity and privacy
- Assurance of end-to-end performance
- 最难的是第 7 个问题、performance 的问题

14. Assurance of At-least-once Delivery
    - RTT (Round-trip time) to_time + process_time + back_time (ack)
    - 问题是：如何确定一个包丢失了
    - 可以通过 timeout 来确定（但是需要首先知道 RTT）
    - nonce：一个随机数、用来标记这个包是第几个包（网络中常用的方法）（唯一用来标识网络中的包）
    - Dilemma 是不知道是哪个阶段丢的包，可能是 ack 的时候或者是发送的时候丢的
    - Try limit times before returning error to app(方法即是每隔一段 timeout 就重新发一个 data 包)
    - 所以他不能保证 No assurance for no-duplication （就是有可能会重复发送）（at least once）
15. 怎么设置这个 timeout 呢?
    - 太长、要长时间才能监测丢包，性能变差
    - 太短、会导致误判
    - 首先不能设置 fixed timer，（有一个路由器的例子）因为没有要到时间就会轮询、然后轮询了之后就更要不到时间、因为会大量阻塞，最后导致服务器崩了
    - 还有一个固定的收邮件的 timer、也会固定导致阻塞
16. 所以用 adaptive timer
    - E.g., adjust by currently observed RTT, set timer to 150%（比如可以简单地设置为 RTT 的 150%）
    - 可以用一个 adaptive timer，比如说根据 RTT 来设置这个时间，或者根据指数增长的方式来设置这个时间
    - 当前 linux 的设置是通过一个不断更新的 avg（存在权重） 和 dev（标准差） 来进行设置的 timeout = avg + 4 * dev（经验数据）
    - 还有一个是 nak(Negative AcKnowledgment)，就是当收到几个包的时候，receiver 会告诉 sender 哪些包丢了，然后对方会重新发这个包，这样 sender 就不需要对每个包维护一个 timer 了、只需要 sender 维护一个全局的 timer（但是 NAK 也会丢包）
    - 只要 at least once、就要有 timer（timeout）
17. Exponentially Weighted Moving Average（具体方法）
    - Estimate both the average rtt_avg and the deviation rtt_dev
    - Procedure calc_rtt(rtt_sample)
    - rtt*avg* = *a*rtt*sample* + *(1-a)_rtt_avg; /* a = 1/8 _/
    - dev = absolute(rtt_sample – rtt_avg);
    - rtt*dev* = *b*dev + (1-b)*rtt_dev; /* b = 1/4 _/
    - Procedure calc_timeout(rtt_avg, rtt_dev)
    - Timeout = rtt_avg + 4*rtt_dev
18. SNTP 里面会有一个 go away 的设置、即如果超过了这个时间、就会让他不要再发包了
19. Assurance of At-most-once Delivery

- 最基本的实现：receiver 收到一个 nonce 相同的包、就不处理
- Maintains a table of nonce at the receiving side
- 但是这样就会让这个 table 无限增长，导致搜索开销变大和 tombstones 的问题
- 另一个方法：通过幂等操作来解决这个问题，允许多次处理同一个包（即一个操作多次执行和一次执行的效果是一样的）（让 application 容忍重新发包）
- 第三个方法：（Duplicate Suppression）还可以设置一个最大的 sequence number（比如上一个接受到的包）、小于这个 number 的包就不接受、大于这个 number 的包就接受，但是这个 old nonce 也会变成一个 tomstone（但是要保证这个 nounce 是递增的、而且不会乱序出现）
- 第四个方法：每次用不同的端口处理请求（一旦服务完就关掉、但是端口是有限的、所以还是需要复用）、但是旧的端口就会变成一个 tombstone

20. Accept the possibility of making a mistake

    E.g., if sender always gives up after five RTT (cannot ensure at-least-once), then receiver can safely discard nonces that are older than five RTT
    It is possible that a packet finally shows up after long delay (solution: wait longer time)

    - 比如 sender 会在 5 个 RTT 之后放弃，那么 receiver 就可以安全地丢弃 5 个 RTT 之前的包，节省了空间

21. Receiver crashes and restarts: lose the table

    - 实现 At-most-once 会比较难、需要更多的资源，而且 crash 了之后因为 table 在 cache 里、会丢失
      One solution is to use a new port number each time the system restarts
      Another is to ignore all packets until the number of RTT has passed since restarting, if sender tries limit times

22. Idempotent could be another choice

    Ensure that the effects of multiple execution equals execution once

    - 所以幂等操作是一个比较好的选择

23. Assurance of Data Integrity

    - 使用 checksum（但是还是有很小的概率，但是不用考虑）（防止数据被改）（在 end-to-end 层添加）

24. Segments and Reassembly of Long Messages（顺序问题）

    - Bridge the difference between message and MTU
    - Message length: determined by application
    - MTU: determined by network（最大传输单元）
    - Segment contains ID for where it fits
    - E.g., "message 914, segment 3 of 7"
    - 需要有一个 buffer、然后填这个 buffer
    - 需要进行拆包
    - out of order 的时候、会有不同的方法解决这个问题（各有优劣）
    - 方法 1：只顺序接受包
    - 方法 2：等待顺序位置的包到了之后、再把这个包放到 buffer 里面（问题是：网络出问题的时候、需要维护一个很大的 buffer，收到之后才能 free、有资源消耗大）
    - 方法 3：combine the two methods above（buffer 满了、就把 buffer 清掉、全部重传）

- 方法 4：当发现一个包没收到的时候、主动返回去要这个包，发一个 NAK 的消息
- 这个 buffer 是 os buffer，所以不能允许一个 app 的占用的 buffer 空间太大、如果 NAK causing duplicates，stop NAKs
- TCP 基于 ACK 的重传（这个是基于超时的、其实是慢一点的），而不是 NAK （NAK 会快一点）

25. Assurance of Jitter Control
    - 相当于看视频的时候的提前缓冲，来避免卡顿，因为看到的是缓存好的，而新来的数据有快有慢地来是感受不到的
    - 公式：（最长时延 - 最短时延）/ 平均时延 = segment buffer 的大小
26. Assurance of Authenticity and Privacy
    - 通过证书机制
    - 加密：公钥私钥（非对称加密）（用公钥加密的东西，只有私钥能解开）（用私钥加密的东西，只有公钥能解开）

20 TCP Congestion Control & DNS

1. End-to-end Performance
    - lock-step protocol:收到 ack 之后再发下一个包，但是问题就是、网络时延会变成一个瓶颈（中间有大量的带宽）
    - 所以引入了 Multi-segment message，可以一次先发多个包（Overlapping Transmissions）（Pipeline style）
    - 但是 Overlapping Transmissions 的时候会有一个问题、需要考虑丢包的时候怎么办
    - 性能由 receiver 收包的能力和网络带宽决定（就是每次发多少个包、停下来等一等）（如果 sender 很慢、receiver 很快、那么就没关系）
    - 所以引入了 fixed window（这个 window 的大小是 receiver 给的，例子里是每次收发 3 个包）
    - 然后为了提升性能、引入了 sliding window（这个 window 会根据目前收发到了多少个包进行滑动）（就能缩短 receiver 的 idle 时间，减少浪费的带宽）
2. 有一个解决丢包的例子
    - 因为 2 号包丢了、所以不能滑到 3 号包的 window 那边
    - timeout 之后会重发 2 号包
    - 所以这个 TCP 解决丢包的方案是比较保守的、会等待 2 号包的 ack 回来之后、再继续发包
3. Sliding Window Size 的大小需要合理进行设置
    - 如果 window 太小了、就造成很长的 idle 时间
    - 如果 window 太大了、就会造成网络拥堵
    - window size ≥ round-trip time（一个来回的时间） × bottleneck data rate（瓶颈的带宽）
    - 500KBps（receive） * 70ms （RTT）= 35KB = 70 segments（a segment is 0.5KB）
    - 但是单单根据这个公式得到的还是不准确的、因为其中可能还会有其他的瓶颈、如果传得太快、就会发送拥堵
    - 而且这个不知道这个瓶颈带宽是多大、而且会可能实时变化
    - 为了解决这个问题、就有了后面的 TCP Congestion Control
4. TCP Congestion Control

- 大量的包出现在网络中、会导致网络拥堵
- 需要既涉及到 network layer 也涉及到 end-to-end layer
- 因为要排队、所以会需要把包存在队列里面，会存在网络资源的浪费（放不下之后、就会丢包，也可能会处理超时的包、所以会浪费）

5. Network layer
    - Directly experiences the congestion（直接感受到拥堵，比如路由器、来不及处理了、就要丢包、有两种选择、一种是丢了、这样端到端也可以发现 ack 丢了，但是 congestion 从发生到被感受到就有很长的时间）
    - Ultimately determine what to do with the excess packets
6. End-to-end layer
    - Control to reduce the sending rate, is the most effective way
7. 所以为了 congestion control，也要控制 window size 的最大值
    - window size ≤ min(RTT x bottleneck data rate, Receiver buffer)
    - 但是 bottleneck data rate 是动态的、所以问题在于怎么找到这个值
    - 节点只能感受到丢包、所以需要通过丢包来感知网络的拥堵
8. congestion control basic idea
    - Increase congestion window slowly（一开始缓缓增加）
    - If no drops -> no congestion yet
    - If a drop occurs -> decrease congestion window quickly（一旦发生了丢包、就要快速减小）
    - 缓慢地增加、迅速地下降、很保守的方法
    - 还需要考虑怎么保证 fairness 的问题
9. 有一个 AIMD 的方法（Additive Increase, Multiplicative Decrease）
    - Every RTT:
    - No drop: cwnd = cwnd + 1
    - A drop: cwnd = cwnd / 2
10. 但是有很多问题
    - 比如一开始的一段时间、速度巨慢
    - 解决方式：只有在第一阶段用指数的方式增长（slow start）（Congestion_window ← 2 * Congestion_window）
    - 这里 ppt 上有一个增长的示意图
11. duplicate ack received 是实际中引入的、receiver 当很多时间没有收到一个包的时候、会重发一个已经发过的 ack、这样主动地告知 sender 出问题了，就能标识网络问题
    - 一旦收到了 duplicate ack，就会减半 cwnd（cwnd = cwnd / 2）
    - 然后如果有一段时间 expires stop sending 之后（就是超时之后）、会重新从头开始 slow start（因为发生丢包了、说明有很严重的问题产生）
    - 但是问题还是有、就是图像上会有区域的带宽没有充分利用
    - 所以有了 DC-TCP 这些改进的方法（贴近最优带宽进行调整）、比如说针对数据中心的场景进行使用（针对不同场景有不同优化）（数据中心带宽很高、而且网路情况很稳定）
12. AIMD Leads to Efficiency and Fairness

    - 就是会先都腰斩变成一半、然后再每个都加 1、然后最后会收敛到这个 y=x 的直线上面
    - 这样不断调整就会调整到中间的点，然后来回震荡（因为落到了中间的线上）
13. 为什么不用减法？因为就没法自动保证 fairness
14. 丢包不一定是由于 congestion 导致的、比如在不稳定的无线网里、由于信号导致的丢包、我们甚至应该加快发包速度
    - 所以传统的 TCP 直接应用在无线网上的时候、会导致恶性循环(动不动就退到 0，所以需要重新设计)
15. Weakness of TCP
    - If routers have too much buffering, causes long delays （性能，延迟）
    - Packet loss is not always caused by congestion（丢包的原因不一定是拥堵）
    - Consider wireless network: if losing packet, sender may send faster instead
    - TCP does not perform well in datacenters（在数据中心里表现不好，因为数据中心的网络是很稳定的，而且带宽很高，时延很低）
    - High bandwidth, low delay situations
    - TCP has a bias against long RTTs
    - Throughput inversely proportionally to RTT
    - Consider when sending packets really far away vs really close
    - Assumes cooperating sources, which is not always a good assumption
16. 总结：Congestion window is adapted by the congestion control protocol to ensure efficiency and fairness
17. Domain Name Service
    - 需要从 host names 转换成 ip 地址（计算机通过 ip 访问）
    - router 不知道怎么发包到域名
18. 为什么不只用 ip
    - 首先不够用户友好
    - 其次 ip 是动态的、会进行变化（地点变化、服务器迁移）
19. Questions on DNS
    - 一个 name 可以有多个 ip address（这样就可以选一个近的、就会更快）
    - 一个 ip 也可以有多个 name（这样就可以有多个域名指向一个 ip、比如说一个网站有多个域名）（server consolidation）（比如说网站使用不频繁的时候）
    - ip-name 的映射是动态的、可以变化（用户无感知）
20. Look-up Algorithm
    - 一开始是存在每个机器里的 txt “hosts.txt” 里、但是不能 scale
    - 所以就有了 DNS server，一开始叫做 BIND（Berkeley Internet Name Domain）
21. Distributing Responsibility
    - 分散：减轻压力
    - 结构化：也能和现实中的企业和组织结构相对应
    - 需要对 name 域名进行结构化、做分层 hierarchy 的结构，然后按层去寻找
22. Name Servers

- The root zone（根域名）
  Maintained by ICANN, non-profit（仅负责一级域名）（数量不用特别多）
- The ".com" zone
  Maintained by VeriSign, add for money
- The ".sjtu.edu.cn" zone
  Maintained by SJTU

23. 它是从后往前解析的
24. Context in DNS
    - Names in DNS are global (context-free)
      - A hostname means the same thing everywhere in DNS
    - Actually, it should be "ipads.se.sjtu.edu.cn."（只是被省略了最后的 .）
      - A hostname is a list of domain names concatenated with dots
      - The root domain is unnamed, i.e., "." + blank
25. Fault Tolerant
    - Each zone can have multiple name servers （为了容错，每个域名都会有多个 name server）
      - A delegation usually contains a list of name servers
      - If one name server is down, others can be used
26. 下面讲了如何 Three Enhancements on Look-up Algorithm
27. 方法 1：可以指定地绑定和搭建 dns
    - 通过修改 hosts 文件
    - 设置/etc/resolv.conf，指定 dns server，如果找到了对应的、就直接从这个 dns server 里返回、如果没有找到、就会去找其他的 dns server
    - 好处是什么呢？既可以自己主动地去改动一些指定的域名、也可以在小范围内增强对域名访问的控制力（监管管理，实验室内部）
28. 方法 2：recursion 的方式的通信性能会更好一点、因为服务器之间的网络稳定、性能会更好
    - 但是要求就是对 root 的要求太高了
    - 而且是需要服务器维护有状态的（root 挂了就没了）
    - 如果 non-recursion、服务器就不需要维护状态
    - 实际中如何利用他们两个的优点呢?
    - 可以使用 caching
29. 方法 3：caching
    - 比如 sjtu 的服务器、不仅会解析 sjtu 的域名、还会解析其他的域名
    - DNS clients and name servers keep a cache of names
      - Your browser will not do two look-ups for one address(服务器 cache 这个映射，就不用每次都去找一次 dns server 了)
      - 电脑上自己也会有 cache
    - Cache has expire time limit（缓存有一个过期时间）
      - Controlled by a time-to-live parameter in the response itself
      - E.g., SJTU sets the TTL of www.sjtu.edu.cn to 24h

    - TTL (Time To Live)
        - Long TTL VS. short TTL
        - 如果太长了、就会体验到错误（比如个人网站部署的服务器迁移更新了，怎么让用户感受不到服务中断，所以需要留 24h 再关掉、因为 dns 的缓存是 24h）
30. 全世界也有很多 root dns
31. sjtu 也需要对内和对外的 dns server
32. 找第一个的地址的时候会比较麻烦、因为还没上网、所以需要口口相传，或者电子邮件
33. Comparing Hostname & Filename（比较 hostname 和 filename）
    - File-name -> inode number
    - Host-name -> IP address
    - File-name and host-name are hierarchical; inode num and IP addr. are plane
    - They are both not a part of the object（他们都不是锁映射的的一部分）
        - File-name is not a part of a file (stored in directory)
        - Host-name is not a part of a website (stored on name server)
    - Name and value binding （多对多绑定的关系）
        - File: 1-name -> N-values (no，没有意义，或者备份、版本管理); N-name -> 1-value (yes，hard link)
        - DNS: 1-name -> N-values (yes，加速和容错); N-name -> 1-value (yes，website consolidation，服务器聚合)
34. 貌似最后还讲了一点东西，dns 就是类似于 metadata server，单机的时候会有瓶颈、所以可以扩展几个 dns server，然后通过负载均衡的方式来进行访问

21 P2P Network

1. Good Points on DNS Design
    - Global names (assuming same root servers)
        - No need to specific a context
        - DNS has no trouble generating unique names
        - The name can also be user-friendly
    - Scalable in performance （性能好）
        - Simplicity: look-up is simple and can be done by a PC（lookup table）
        - Caching: reduce number of total queries
        - Delegation: many name severs handle lookups （很多服务器可以处理查询）
    - Scalable in management
        - Each zone makes its own policy decision on binding
        - Hierarchy is great here
    - Fault tolerant
        - If one name server breaks, other will still work
        - Duplicated name server for a same zone
2. Bad Points on DNS Design
    - Policy

        - Who should control the root zone, .com zone, etc.? Governments? （这个 root zone 应该由谁来控制？）
    - Significant load on root servers
        - Many DNS clients starts by talking to root server
        - Many queries for non-existent names, becomes a DoS attack
    - Security
        - How does a client know if the response is correct?
        - How does VeriSign know "change Amazon.com IP" is legal?
3. 怎么解决域名服务器逐层 cache miss 的问题（转化为 DoS 攻击）
    - 比如输入一个不存在的域名
    - 可能 root 更新的速度比 sjtu 的速度要快，所以可能会误报
    - 用分层定时扩散域名的方式（sjtu 定时获取）（root 不知道 sjtu 在哪、但是 sjtu 知道 root 在哪）（但是缺点是可能新域名要过一段时间才能获取到）
4. Security：使用证书机制保证 dns 服务器的安全
5. Behind the DNS Design（下面讲了原理，naming schema）
6. Naming for Modularity
    - Retrieval（检索）
    - Sharing（比如说硬链接）
    - Hiding（虚拟地址和实际地址解耦，保证了安全性）
    - User-friendly identifiers
    - Indirection:与实际位置解耦
7. Software uses these names in an obvious way
   E.g., memory addresses
   Hardware modules connected to a bus
8. 比如：EAX - 是一个寄存器的名字
    - 所有机器上都有 EAX，而且叫这个名字的寄存器有很多个（实际映射到很多不同个的隐藏的寄存器上）
    - 所以可以并发地去做一些操作
    - hiding
    - indirection（所有机器上都有 EAX，但是代码不用关心这个 EAX 是什么，只需要知道这个 EAX 就行）
9. 或者 phone number 的虚拟号码、也是保护隐私（indirection）
10. naming model 里的 context 有些是 global 的、有些是 local 的，比如针对说一块磁盘（这张图很好）（或者域名的 context 就是全球、或者没有 context）
11. bind 和 unbind，unbind 很多时候现在做得不好（电话卡）
    - Binding – A mapping from a name to value
    - Unbind is to delete the mapping
    - A name mapping algorithm resolves a name
12. Naming Context
    - Type-1: context and name are separated

    - E.g., inode number's context is the file system
    - Type-2: context is part of the name
    - E.g., xiayubin@sjtu.edu.cn
    - Name spaces with only one possible context are called universal name spaces
    - Example: credit card number, UUID, email address
    - Embedded in name itself
      cse@sjtu.edu.cn:
      Name = "cse"
      Context = "sjtu.edu.cn"
    - Taken from environment (Dynamic)
      Unix cmd: "rm foo":
      Name = "foo", context is current dir
13. name mapping algorithm 本质上就是查表 Table lookup
14. recursive lookup 是递归的，multiple lookup 是平行的（比如说$PATH，有很多个、但是不是递归的关系）
15. FAQ of Naming Scheme
    - What is the syntax of names? 比如说网址里的.分隔符
    - What are the possible value?
    - What context is used to resolve names?
    - Who specifies the context?
    - Is a particular name global (context-free) or local?
16. 这里说的 5-4-5 范式是什么意思?
17. content distribution
    - 如果每次都要去 b 站访问视频，会有很大的带宽消耗，所以需要 cache 到 sjtu 的服务器上，然后客户再从 sjtu 的服务器上获取
    - 相当于就是分布式的 cache，比如 b 站服务器把视频主动推送到 sjtu 的服务器上、然后避免对主视频服务器的多次访问（缓解了压力）
18. content distribution network（CDN）
    - Network of computers that replicate content across the Internet
      - Bringing content closer to requests
      - End users only use local (not shared) network capacity
    - Content providers actively pushes data into the network
      - Improve performance and reduce costs
19. 访问时在选择目标服务器的时候
    - 一开始是通过 http 重定向在进行，但是这样会有很多的 overhead（额外的 round trips）
    - 然后是通过 dns 来选择、是选择最近的（避免了 redirect）（缺点是通过 ip adress 来找这个 dns 服务器，但是不一定是最快）
    - 然后演化成了 akamai 的形式、逐层地找 dns、直到找到最近的 cdn 服务器，最后从最近的 akamai 服务器获取内容数据
    - cdn 提供商会从 content provider 主动获取数据、然后再 cdn 的服务器上缓存，然后再把内容数据发送到对应的 end user 的机器上

- 用户一定是从最近的 nearby cdn server 上获取的内容、而不是从 content provider 上获取的
- 好处是减少了带宽成本（如果每次用自己的服务器、成本巨大）

20. 下面讲的是 P2P
    - 就是相当于如果把 cdn 放在家里
    - 中心化的基础设施会有一些缺点：比如说单点故障、性能瓶颈
    - P2P 是一种去中心化的方式，没有中心化的节点
21. Usage Model: Cooperative
    - User downloads file from someone using simple user interface
    - While downloading, BitTorrent serves file also to others
    - BitTorrent keeps running for a little while after download completes
22. BitTorrent 有三个角色
    - Tracker：What peer serves which parts of a file
    - Seeder: Own the whole file （是一个特殊的 peer）
    - Peer: Turn a seeder once has 100% of a file (有一部分文件、100%文件的时候就变成了 seeder)
23. 首先会需要把 .torrent file 发到一个中心化的 web server 上（根节点，公共知道的地方）
    - Publisher a .torrent file on a Web server
        - URL of tracker, file name, length, SHA1s of data blocks (64-512Kbyte)
    - Tracker
        - Organizes a swarm of peers (who has what block?)
    - Seeder posts the URL for .torrent with tracker （把自己的文件信息告诉 tracker）
        - Seeder must have complete copy of file
    - Peer asks tracker for list of peers to download from （peer 会问 tracker 有哪些 peer 可以下载）
        - Tracker returns list with random selection of peers （tracker 会返回一个随机的 peer 列表）
    - Peers contact peers to learn what parts of the file they have etc.（peer 会联系其他 peer，了解他们有什么文件）
        - Download from other peers
    - 这些信息可能马上会变动（比如下完了马上关掉）
24. 这个去中心化建立在中心化的基础上（traker）
    - tracker 起到了中心化的作用
    - 所有这个网站就保证了定期发布了 tracker 的信息
25. 这个 bitTorrent 有不同的下载策略
    - Random
    - Rarest first（比如从最少的资源开始下载，这样可以保证整个文件能够下载下来）
    - Strict
    - Parallel
26. bitTorrent 的缺点就是 rely on tracker

27. 所以引入了一个 DHT: Distributed hash table
    - 即在分布式下的 hash table
    - 有 put 和 get
28. Chord Properties
    - Efficient: O(log(N)) messages per lookup
    - N is the total number of servers
    - Scalable: O(log(N)) state per node
    - Robust: survives massive failures
29. Chord IDs
    - SHA-1 is a hash function
    - Key identifier = SHA-1(key)
    - Node identifier = SHA-1(IP address)
    - 这个 key id 和 node id 是在同一个 value space
30. Consistent Hashing
    - 就是这个 node 对应负责小于等于这个 node id 的 key
    - lookup 的时候、node 会一个一个往前、然后直到找到一个 node 能够负责这个 key（O（n））
    - 然后可以用一个"Finger Table" 的优化、让这个查找更快（二分），然后每个节点也不需要保存所有的邻居节点、只需要保存比较少的邻居节点 O（log（n））
    - 如果有 failure 之后，会有一个问题：就算例子里的 K90 其实最后是保存在 N113 里的，但是 N120 不知道，所以会从 N120 继续往后找（相当于跳过了 N113，就找不到 K90 了）
    - 可以使用 successer lists 来解决这个问题（每个节点保存自己的后继节点）
    - 这时候最简单的方法就是顺序，所以解决方法是合二为一（同时维护 finger table 和 successer lists）
    - list 要多长：不会有人退出的话、可以短一点、如果频繁变化、可以长一点
31. 这个初始化的例子里、新加入节点之后、原来节点里的内容也不用删除，因为查找对应 key 的时候自己的负载也不会变大
    - 比如说一开始加一个 n36
    - 第一步、把 n40 上面的小于等于 36 的 key 都给 n36
    - 然后再把 n25 的指针指向 n36
    - 这里 n40 中 k30 还是在的
32. 做负载均衡的时候、可以引入虚拟节点的概念、然后和实际节点做映射，让负载均衡做得更均匀
    - Load balance could be a problem Some nodes have most value
    - One physical node can have multiple virtual nodes
    - More virtual nodes, better load balance

22 The distributed (and parallel) programming: it’s all about scalability

1. 怎么去加快计算呢，方法就是 Parallelism

2. 一开始是硬件上的 pipeline，但是单核上的频率有瓶颈、因为超标量和主频的提升会停滞。所以第一个方法变成了多核
3. 在多核情况下、会有一个 Cache coherence protocols，保证内存的一致性
    - 一个是 Snoop-based cache，还有一个方法是用一个来统一
4. 但是问题是不能 scaling、因为会涉及到要 snoop 很多个核（gpu 是没有 cache coherence 的）
5. 第二个方法是通过加更多个 alu 单元（本质区别是和更多的硬件空间留给了 alu）
6. New ISA: SIMD processing：用了 SIMD 之后、需要额外写很多代码
    - 每次可以做 8 个 4bit 操作
    - why not 8X faster? 因为不仅仅是做计算、还需要做访存（访存速度没有那么快）
7. 访存有两个限制
    - 一个是访存的延迟
    - 一个是访存的带宽
    - Both factors matters ， 取决于实际应用的场景和需求
8. 用 SIMD 优化之后、cpu 的性能就受制于访存上了（可以通过计算）

LEC 22: Distributed Computing: Intro

- 人工智能的发展导致需要大量的计算资源 也就必须得扩展为分布式计算
- AI 训练计算所需要的性能是非常好计算的
    - 一个神经网络层 W 是 m*k 的矩阵 X 是 k*B 的矩阵 B 为 batch size k 为输入维度 m 为输出维度 那么计算 W*X 的时间复杂度为 (2k-1)*m*B
    - backward path（反向传播）需要计算 dW 和 dX 时间复杂度为(2m-1)*k*B 和(2B-1)*k*m 都可以约为 2*参数数量*Batch Size
    - 最终的结论就是 AI 模型训练总的时间复杂度为 6*参数数量*Batch Size
    - 对于 GPT 来说 有 1.76trillion 个参数 一张 A100 显卡的性能是 19.5TFLOPS 那么训练一个 GPT 需要 30s 进行一次迭代 非常的耗时
- 我们最终的目的是使用足够的算力进行分布式训练 接下来将先从单个计算设备开始讲起

Computing Device

- 单个设备提升算力的根本就是并行

CPU

- 单核 CPU 系统 算力上限取决于 Clock Rate 一个时钟周期最多做一次指令 （当然也有使用 pipeline 和超标量（一次性做 4 条或者 8 条指令，本质是预测，前提是下一条指令和上一条指令没有关系）的方法来提高单核的算力）然而硬件的主频基本上因为散热问题已经到了极限
- 多核 CPU 系统 算力上限取决于核心数 然而和分布式类似 最恶心的还是 cache 中的 coherence 问题
- 另一种类似多核的方法是增加每个核的 ALU 数量 即 SIMD（Single Instruction Multiple Data）的思想 一条指令处理多个数据
    - 需要新的指令集（比如 Intel 的 AVX）增加了代码的复杂度
    - 性能增益不可能达到理论的倍增 因为指令可能有依赖
    - 有时候访存也会成为性能瓶颈 访存的时间 = Latency + Payload / Bandwidth

    - Roofline Model：刻画算力与带宽的关系图 y 轴是算力 单位是 GFLOP/s 即每秒多少次浮点运算 x 轴是应用利用内存的效率 单位是 Flop/Byte 即读取一个字节数据可以进行多少次运算 于是斜率就是应用对于带宽的需求 单位为 Byte/s 于是设备在图中表示为两条线 一条代表算力的水平线与一条代表带宽的斜线 应用的 OI（Operational Intensity）所落在哪条线 就说明哪个因素是瓶颈

GPU

- 也是两种方式提升算力：增加核心数和 SIMD
- GPU 的 ALU 数量远远多于 CPU 因为 GPU 没有 cache coherence
- SIMD 难以实现分支条件 但是可以通过 mask 来实现 也就是执行所有分支 但是写回的数据会用条件的 mask 来选择 不过还是会造成很多无用的计算 而且编程变得很复杂
- 于是 NVIDIA 实现了 SIMT（Single Instruction Multiple Threads）：（没听懂 说是超纲了）应该就是相当于用户指定某一个线程执行某一个任务 比如指定第 i 个线程执行第 i 位的相加

TPU

- Tensor Core：专门用于矩阵运算的核心 替换掉了原来的 ALU 用于加速矩阵运算
- TPUs（Tensor Processing Units）：谷歌的专门用于矩阵运算的芯片

Summary

- 单个设备的算力由于种种限制没法无限提高（比如主频是因为散热问题 多核是因为芯片面积问题等）连老黄的显卡的提升也只是因为使用了 tensor core 或是增加了核心数
- 于是就需要分布式计算来提高算力 大部分情况下使用一个开源的分布式计算框架足以 接下来我们介绍一个经典的分布式计算框架：MapReduce

23 The distributed (and parallel) programming on a single device & MapReduce

1. mapreduce 是一种分布式的计算框架
2. 超标量用于单核，但也有上限（pipeline）
3. 可以用 SIMD，但是传统的 cpu 不支持、而且需要写额外的代码
4. SIMD 之后、继续增加的话就是用多核（还有多个 ALU），（不能无限增加数量是因为硬件的数目和空间是有限制的）
5. 如果还没从内存中 load 出来的话、cpu 就会空转
6. roofline model 就能刻画一个设备时受制于算力还是访存（y 轴代表每秒需要计算多少浮点数）（横竖线时设备的计算能力）
7. 怎么刻画访存需求：访存本质上是间接需求（好的应用是每次 load 能做很多次计算、而不是每次计算都需要 load 内存）
8. x 轴刻画的是 operation intensity （越大、越能利用算力、越小、访存越多）
9. 斜线即代表了它带宽的需求（斜率）
10. 左边 intensity 和斜线相交、就是访存 bottleneck、在右边和 peak flops 相交、就是算力 bottleneck（给出应用的优化方向 application intensity）（比如说可以优化 gpu 算力、或者是将 intensity 竖线右移）
11. 因为 gpu 上面没有 cache coherence，也没有预测分支，也没有预取的硬件空间、所以可以承载更多个 SIMD ALU，所以 ALU 能放更多

12. 但是 SIMD 的操作只是针对底层硬件的，比较原始（而且是 vector 中批量做的）
13. 对于 relu 这种函数怎么做优化呢、可以先都做计算、然后如果小于零那么就让结果不生效（mask 方法）
    - 本质上是 branch prediction（每个 branch 都做一遍）
    - ppt 上的例子是：比如在分支中、先全部做 True 的情况、然后再把对应的情况写入寄存器、然后再做 False 的情况、再做写入寄存器（实现了 conditional branch 中的减少计算、这样只需要做两次）
    - 但是这种方法还是会有 50% 的计算浪费
14. cuda 是 nvidia 的 SMID 的框架
    - 是一种 SIMT，在 SIMD 上抽象了一个更高层的概念
    - 然后就不需要写很多 SIMD 中的重复的代码了，比如 if 里的 mask 操作
    - 每个 thread 在同一个时间都只执行同一条指令
    - 有了这个方法之后、就能用 CUDA 去写多 kernel 上的代码了
    - 矩阵乘法的硬化是会有很大的效能提升的
    - 性能提升主要在于 tensor core
15. 主频的提升是有上限的、因为会有很多的热量产生
16. 但是单卡还是不够、所以需要分布式
    - 存在很多分布式计算框架
    - batch processing（spark、hadoop）
17. batch processing
    - 场景：谷歌要统计很多网站每天的点击量
    - 传统的是在 log 中用命令行去统计，但是它是单线程的，很难 scale
    - 所以可以手动做一个任务划分、每台机器做一个 log 的统计，然后最后再做汇总
18. rpc 不够高效、因为有集群
19. mapReduce
    - map：把一个函数应用到很多个数据上（因为 map 这个事情非常适合并行化）
    - reduce：把 map 的结果做一个累加（必须等 map 全部做完）
    - 这里假设了函数程序不会有 side effect、所以如果出错了之后、直接重做就可以了
20. 这个 reduce 其实是根据 key 做 reduce 的
    - 这样就可以让 reduce 也具有并行性
    - 用 partition 的方式来类似地去做 reduce
    - 有了 mapreduce 之后、我们只需要考虑 functionality 就可以了
21. 例子：统计页面中出现了多少次每个单词
22. 只需要把代码直接给 mapReduce runtime 就可以了
23. MapReduce 执行流
    - 就是把原始数据分为不同的 shard 来进行（64MB）
    - master 负责启动所有的 mapper 和协调 reduce worker 什么时候开始执行
    - 对于用户是不可见的

- 第一步：把大的文件分为很多小的 chunks（不能太小也不能太大）（太大的话在不同的 chunkserver 上、需要从很多个 server 上面拉这个数据）（选择了 64MB 的大小、因为 gfs 的 chunk 大小是 64MB，就能保证一次 rpc 就可以了）
- 第二步：用 master 起所有的 map worker 和 reduce worker（一台机器上可以起多个 map worker 和 reduce worker，比如 reduce worker 不做事情的时候其实不怎么耗费资源）
- 第三步： map task（它的所有中间结果都是先放在内存里、这样更快、然后再异步的方式去刷回到磁盘）
- 第四步：这个 intermediate files 需要保证每个 reduce worker 负责的 key value 要放在一起、不然的话会有很多的磁盘上的随机访问、就会降低性能（会把文件做 partition，如果是 reducer1，就写到 partition1 里，每个 partition 对应于一个 reducer）
- 第五步：mapper 结束之后、会告诉 master 可以启动 reducer 了，然后 master 会指定哪些 reducer 去读哪些 partition。这里的先 sort 是为了保证 reducer 在读的时候、一次可以把一个 key 的所有 value 都读完（如果在本地排序整个文件、会有很多 overhead、所以有一个优化，就是类似于归并排序，每个 map worker 会把一个 partition 的 key 先做一个排序，然后 reducer 做合并的时候就用归并排序的方式）
- 第六步：做 reduce，产出最终 output file
- 第七步：把结果返回给用户

24 Distributed Computing frameworks MapReduce, computation graph & Distributed training

1. mapReduce 的优化：首先是要做 fault tolerance
    - 一种是 worker failure
    - 一种是 master failure
2. 有两个让 mapReduce 很好 handle failure 的原因
    - 因为这个 map or reduce 的 function 没有 side effect，所以可以直接重做
    - 基于一个可靠的服务 GFS
3. worker failure
    - 第一个是 detection：监测心跳包（或者是 timeout 时间到了）
    - 第二个是 recovery：重做这个任务，会给这个 worker reset 到初始状态、或者直接把这个 job 给其他的 worker（re-execution）
4. master failure
    - 因为这个 job 往往会需要很长的时间、比如 8 个小时。所以直接重做不太好
    - 所以 master 会把自己的状态定期地持久化到 GFS 中、然后如果 master 挂了、就会从这个状态中恢复
    - 它是一个 periodical 的 checkpoint（所以就不需要重做所有的操作了）（但是做 checkpoint 还是会有性能开销）
5. 用户给 mapReduce 的函数，如果会有第三方库、有可能会触发这些第三方库的 bug
    - 一种是 report 给 developer、但是 developer 不一定写的这个库
    - 后来发现这些很多 bug 是由于一些特点和报错、比如会导致 segmentation fault
    - 如果出了这些错、那就把对应的 input 数据给直接丢掉不计算了
6. Optimization for Locality

    - 因为带宽和网速不够快，所以需要尽量减少网络的传输
    - 当时的计算和存储是在同一个机器上的、所以可以直接从本地读取数据然后在本地计算
7. stragglers 是那些跑得特别慢的机器 （Refinement: redundant execution）
    - 因为一个机器上可能不仅仅跑了 mapReduce 的进程、所以会影响性能
    - 或者 cpu 没有 cache（由于配置错误）
    - 所以会出现如果只有一个 mapper 没有做完的话、那么就需要等待这个 mapper 做完才能做 reduce
    - 解决方法：做冗余，每一个 job 可以分在不同的机器上面去执行，如果这个 map 在任意一个机器上被做完了、那么就认为它做完了。
    - 但是这个在训练的时候不太使用、因为训练模型需要很多计算
8. reducer 0 负责 0-1000，reducer 1 负责 1001-2000
9. Sort 的时候也是分 partition 去做的
10. 但是，MapReduce cannot address all the issues
    - 因为本身这个 map reduce 的函数比较简单（通过迭代，就是没有方法去编写一些比较复杂的例子）
    - 但是就不适用近似实时的计算和场景了
11. start penalty：启动 mapper 的时候需要预热（fork 进程）
12. 可以通过直接把 map 的结果发送给 reducer，而不是先写到磁盘上，来做优化（减少写磁盘的时间）
13. 为了抽象更复杂的计算，就需要 computation graph
14. computation graph
    - 一个节点代表一个数据或计算（要么是一个数据、要么是一个计算）（没有 side effect）
    - 一个边代表数据的依赖关系
    - 一个 job 就可以表示为一个 DAG
    - 因为 DAG 对于容错比较友好、如果一个红色的节点挂了怎么办、那么就可以去它的上游找对应的数据还在不在，然后做对应的恢复
    - 有环就不干净了
    - DAG 还能判别出哪些任务是可以进行并行执行的（没有路径依赖的话）
15. 用 kv store 存储中间状态的数据（内存、不存硬盘、因为硬盘太慢了）
16. 这个 failure 也会有 recursive 的情况
    - 只要回溯找到第一个没有挂的节点，然后就可以从这个节点开始重新计算
17. computation graph 也把系统和底层的算法做了解耦（机器学习）
18. 也可以去做图优化（把几个小的 kernel 合并成一个大的 kernel，好处就是 kernel 的访存效率提高了）（cuda graph）（针对于 gpu 硬件的实现）
19. Graph fusion：将多个节点合并成一个节点 减少通信开销
20. 下面开始讲 distributed training
21. x 是数据、y 是 label（期望值），w 是参数，然后一轮之后就是更新
22. 但是这个计算图的问题是、它没有办法并行、因为全是 dependency
23. 所以分布式训练研究的就是怎么把一个计算图变成一个可以并行化的计算图
24. 有两种主要的并行化方式

- 一种是 data parallelism，就是把数据的 batch 分成很多份，然后每个机器上都做子 batch 的计算，这里的限制就是每个计算图的节点上面都有一个完整的模型、但是对现代的深度学习模型来说、这个模型是很大的、所以这个方法不太适用
- 一种是 model parallelism，就是把模型的计算过程分成很多份，然后每个机器上都做一次计算，然后再把结果合并
- 还有一种是跨 iteration 的 parallelism，就是可以做同步或者异步的更新（讲得会比较少）（现在基本所有大模型都是同步更新出来的）

25. 同步的 data parallelism
    - 图中就显示出了讲输入的数据去做分割，然后每个机器上都做一次计算，然后再把结果合并
    - 最后的梯度需要相加
    - 最后的问题是它什么时候做汇总，因为要 syncronize，所以会有一个 bottleneck（要等所有的都算完）
26. 这个 allreduce 是因为要把汇总的结果给所有人广播、所以叫 all
    - 问题就是怎么把这个汇总的操作做得快
27. 里面最大的问题是需要减少 network overhead
    - Overhead analysis: computation + network（computation 的其实很少）
    - 一个是带宽的 bottleneck
    - 还有一个是瞬时的 concurrent 的 message 的数量（比如 1 万张卡的数据同时给到一个机器）
28. 实现 1：parameter server
    1. Each process pushes the data to the parameter server (PS)
    2. After receiving all data, aggregate the data at the PS
    3. The PS pushes the data back to the processes
    - 好像就是一个简单的模型
    - 对 parameter server 的性能需求（带宽和并发数）和机器数是成正比的 O(N * P)
    - N: the size of the parameters; P: the number of processors
    - Step #1: O(N) at each process, O(N * P) at the PS
    - Step #3: O(N) at each process, O(n * P) at the PS
29. Co-located & sharded PS
    - 每台机器只负责 1/n 份的数据
    - Total communication per-machine： O(N * (P – 1) / P)
    - Total communication rounds: O(1) -> Good!
    - O(P) fan-in -> Bad!
30. fan-in 就是很多机器在同一个时间给同一台机器发很多流量，会造成阻塞
    - Congest control overhead （控制阻塞信息的 overhead）
    - Resource contentions
    - 所以希望同一时间只有一台机器和 x 通信
31. 第二个方法：De-centralized approach for allreduce
    - 做一些协调、让每个机器的发送有一定间隔、而且不在同一瞬时
    - 每个机器做完之后主动通知下一台机器去发

- 然后用类似方法发回信息
- 这个 fan-in 就是 1
- 但是还是有问题、就是每台机器要传的参数量还是很多，而且这个通信的轮数还是很多 O(P * P)

25 Distributed Computing frameworks MapReduce, computation graph & Distributed training

0. 业界实际场景会使用 3D Parallelism 即同时使用下面三种并行化方法
1. naïve decentralized reduce 有个坏处、就是虽然它的 fan-in 是 1，但是它需要传的总数据量很大、是 N * (P – 1) data(就是要传完整的参数)
2. 还有一个是用类似分 shard 的方式管理、但是还是有问题、就是当连接数大的时候、还是会有很多通信和网卡受不了的情况出现
3. 也可以采取一个优化方法 Ring allreduce（就是通过类似一个环的方式去传每个 shard 的数据）、就是每个机器只和邻居相连，然后然后每次传的时候做一个 reduce，然后再把 reduce 完的结果传给下一个邻居。然后最后一个传到的机器就有一个完整的数据。（这样的话每台机器的网络连接数就是 constant 了）（量还是一样的、但是就是每次传的数据不一样）
    - 好处是不需要 coordination、而且每个机器只需要维护邻居的网络链接了
    - 就是每台机器只负责一个 partition
    - 但是问题是其他人没有对应的最终结果。所以需要再做一个 broadcast，然后每个机器都有了最终的结果（训练中的完整的梯度）。
4. 但是由于有些 gpu 计算中的浮点操作（比如加法）是不满足交换律的，所以这个 Ring allreduce 会改变一些计算顺序、所以应用的时候需要注意。
5. Total communication per-machine 总数据量 2 _ (P-1) _ N / P
6. O(1) fan-in
7. 但是上面的 decentralized reduce 还是有问题、因为它还是需要做 k 轮通信
    - Total communication rounds: O(P)
    - Much higher than the parameter server approach (O(1))
8. Double Binary Tree All-Reduce
    - 为什么需要 double binary tree 呢，因为这个单个的 tree，load balance 不好，所以需要两个 tree（因为有些节点是根节点、有些则是叶子节点）
    - Log(p) rounds of communication
    - 所以可以把数据切成一半、然后把一半给叶子节点做、一半给父节点做
    - 这个 fan-in 就只有 O（1）
9. 主要就是三个参数的权衡
    - Communication round
    - Peak node bandwidth
    - Fan-in
10. 上面都是 data parallelism，下面开始讲 model parallelism
    - data parallelism 中，每台机器都需要存储完整的参数 w，但是现在的 llm 太大了
    - 而且现在的 llm 还需要存储优化器、比参数还大

11. 这个 model 的切法就是切分开来这个 w，然后横着切（不同 gpu 放不同层）、就是 pipeline parallelism，纵着切（把一个很大的 w 切到不同的机器上）、就是 tensor parallelism
12. pipeline parallelism
    - 会有一个 bubble 的问题、所以需要 reduce bubble
13. 第一个优化 pipeline parallelism 的方法是 micro-batching，就是把一个 batch 切成很多小的 batch，然后每个小的 batch 都可以并行计算（类似于 cpu pipeline 中的流水线并行）
    - 这个在 forward path 里面还不错
    - 但是在 backward path 里面就不太好了、因为还是需要等到所有的都算完才能做进行计算，所以还是有 bubble
14. Question: what is the overhead of pipeline parallelism?
    - bubble 的时间是 p-1
    - Bubble time fraction: $(p-1)/m$
15. Optimization directions 但是都不大可行
    1. increase m (e.g., increase the batch size) （首先、每个 batch size 不能太小因为需要每个 batch 都保证一定的最小大小、因为如果太小、这个 gpu 的计算效率会很低）（然后这个 batch size 不能太大、因为会导致这个模型的收敛会很慢）
    2. reduce p (reduce the #partitions) （不太现实、因为 gpu 放不下那么多层的参数）
16. 所以有了 tensor parallelism
    - 相当于一层参数放到了不同的机器上面去
    - Partition the parameters of a layer
    - Each partition is deployed on a separate GPU
    - 使用了分块矩阵乘法
    - 涉及了 forward pass
    - 也涉及 backward pass，要算 ∇X 和 ∇W
17. tensor parallelism 和 pipeline parallelism
    - 前者比后者的通信量要多很多
    - 一般现在的卡会先在一个 scala network 上面、这个 network 用 nvlink 连接、带宽差不多在 800GB/s 左右、然后这个 network 里面、用 tensor parallelism
    - 然后在机器之间、会用比较慢的网络、比如 RDMA、是 100GB/s，在跨机器的时候用 pipeline parallelism
18. 3D parallelism 就是把上面三种并行放在一起
    - nvlink
    - data parallelism 用在四台机器的 data replica 中
    - 一台机器内用 tensor parallelism
    - 机器之间用 pipeline parallelism
19. 异步更新
    - 是为了解决这个 struggle 的问题
    - 算完之后、马上把梯度发给下一个人、然后用现有的结果再去做下一轮计算
    - 讲得比较少、因为会影响精度（可能只有在传统的图计算里面有用）

26 Introduction to System Security

1. 一开始的例子、就是说一个商家假扮成一个买家、然后篡改了交易的签名、然后 amazon 就会打钱给他、而不是给原来的商家。
2. Internet makes attacks fast, cheap, and scalable
3. Users often have poor intuition about computer security
4. Security is a negative goal
    - 指目标是一件事情不能发生，所以安全很难判断
    - 比如说有很多可以获取一个文件的方法
5. Why not using fault tolerant techniques for security?
    - 有时候可以 Some security issue can be seen as a kind of fault
    - 但是其中的攻击和 fault 的原因是不一样的
    - 而且这个 attack 导致的 failure 基本是 correlated 的、而单纯的 failure 是 independent 的
    - 所以 No complete solution; we can't always secure every system
6. What are we going to learn?
    - How to model systems in the context of security
    - How we think about and assess risks
    - Techniques for assessing common risks
    - There are things we can do to make systems more secure
    - Know trade-offs
7. Policy: Goals （下面的 CIA），拆开来的原因是因为他们三个相对来说是正交的。
8. Information security goals:
    - Confidentiality: limit who can read data （加密解决）
    - Integrity: limit who can write data
9. Liveness goals:
    - Availability: ensure service keeps operating
10. 拆解目标之后、就可以针对每个目标做对应优化
11. Threat Model: 就是做出 Assumptions
    - What does a threat model look like?
    - 建模之后、就能进一步简化问题
    - 一个假设是只有打开的一个软件是恶意的、其他的都不是恶意的

27 ROP and CFI

1. Authentication: Password
    - 这个攻击的例子是、如果密码的第二部分在内存里的另一页、那么如果前面几位匹配了、那么就会换页、就会慢、如果前面几位不匹配、那么就不会换页、就很快（这样就可以通过时间来判断密码的对错）（时间就变成了 26*8 而不是 26^8）
    - 但是今天这个攻击不太好做、因为要把内存设置成这个样子的话不太方便，但是在当时的 os 里面很容易实现
    - 解决方法：先 hash 或者全部匹配完再返回 true or false

2. Idea: Store Hash of Password
    - 如果密码到 hash 值是一一对应的、那么就可以通过 hash 值来判断密码是否正确（构造）
    - Often called a "rainbow table"
    - 所以为了解决这个问题、需要往密码里加盐（salt）
3. Salting: make the same password have different hash values
    - 就是输入密码的时候、存之前再用一个随机数加到一起哈希
    - 加盐是明文的、但是从根本上还是没有解决这个问题
    - 但是为攻击者加高了很多攻击的代价
    - 因此这个是从工程的角度解决这个安全的问题的
4. Bootstrap Authentication
    - Do not want to continuously authenticate with password for every command
    - 就是为了避免每次操作都要输入密码
    - web 里面是用 cookie 的方式做保存的
    - {username, expiration, H(server_key | username | expiration)}
    - cookie 里面因为用了 server_key 即服务器端的 key 做 hash、所以难以伪造
5. Session Cookies: Strawman 的问题
    - 但是其中会有一个二义性、就是在做字符串拼接然后做 hash 的时候、会有不同的用户名+日期会得到相同的结果字符串（字符串拼接的时候会损失语义）
    - E.g., "Ben" and "22-May-2012" may also be "Ben2" and "2-May-2012"
    - 所以需要用更精确的方式、比如用一个特定的分隔符
6. Phishing Attacks
    - 钓鱼攻击、就是让你输入密码到别人的网站里
    - 比如域名里面只有一个小字符不一样
    - 这个就是密码最大的问题、一旦知道了密码、别人就可以伪造你的身份
7. 下面的 R 是：a random value R
8. Tech 1: challenge-response scheme
    - 如果朴素地做传递密码、会在通信的时候密码被截获，不管这个值是不是被 hash 过了的、都会有这个问题（如果被 hash 过了、之后也可以继续用这个值）
    - 服务器给用户一个随机数、然后客户端用密码和这个随机数做 hash、然后再发给服务器
    - 服务器也有这个随机数、然后也做 hash check、然后比较
    - 这样就不会有密码泄露的问题
    - 但是缺陷是 server 这里需要保存明文
9. Tech 2: use passwords to authenticate the server
    - 这里有两种小的子方法、第一种是 Client chooses Q, sends to server, server computes H(Q + password), replies， Only the authentic server would know your password!
    - 就是用密码去反向验证服务器的真实性（是用来反钓鱼的）
    - 但是真实的系统很少有这么做的，因为可以用同样的一种方式去欺骗 server
    - 第二种是 First, try to log into the server: get the server's challenge R
    - Then, decide you want to test the server: send it the same R

- Send server's reply back to it as response to the original challenge
- 简单概括就是发 server 一个随机数、然后 server 对这个随机数做一个 hash、然后发给 client、然后 client 用这个随机数发给 server、就可以登录了（就是两个方法合起来？）
- 但是这个貌似也是不行的、会变成一个中继，什么都不用干就可以假冒一个用户登录

10. Tech 3: turn offline into online attack
    - 就是加一个验证码、这个验证码是每次都不一样的、所以就不会有中继的问题了
    - 然后服务器还可以记录哪些 ip 是在不停地发送请求（通过日志记录）、有可能会有固定的 ip 发送不同的用户名密码、就可以 ban 掉这个 ip
11. Tech 4: Specific password
    - 用户端：每个服务设置一个不同的密码
12. Tech 5: one-time passwords
    - 就是用一次就不能用，比如手机的验证码
    - 所以会有一个 Hash n 的密码发送策略、然后每次用完之后就会把这个 hash n-1 次后的密码发给 server、然后 server 再做一次 hash，然后发一次、本地的下一次发送、减去一个 1 再发送一次
    - 这个是为了防止攻击者投到中间的 hash 密码
    - 这里还有一个银行用于小的验证码的验证的工具、就是一个不联网的设备、通过初始状态相同、然后同步时间信息、就可以和银行中目前的验证码相同、然后且保证这个验证码不是固定的。然后这里还会需要涉及到银行端做时间上的容错、因为这个时间可能会和设备的时间有微小差异、所以应该在银行端可以计算出一个时间范围内的验证码，都合法。
    - Server stores x = H(H(H(H(...(H(salt+password)))))) = Hn(salt+password)
    - To authenticate, send token=H{n-1}(salt+password)
    - Server verifies that x = H(token), then sets x <- token
    - Google's App-specific password：每个密码只能用一次、然后每次都会有一个新的密码，上一次登录会记录登录的用途
13. Tech 6: bind authentication and request authorization
    - 比如转账这个操作、单单登录之后、使用 cookie ， 不输入密码是不行的、还需要再输入一次密码
14. Tech 7: FIDO: Replace the Password
    - 比如说指纹
    - 把密码放在一个安全的区域、还是用指纹去获取这个密码
    - 最后还是用密码去做登录的验证
    - Three Bindings 是阐述了指纹密码的实现：就是服务器先通过 userID 加上个人的公钥、生成一个东西、然后发给 client、client 用这个东西加上自己的私钥做一个签名、然后发给服务器、服务器用这个签名和自己的公钥做一个验证、然后就可以登录了
    - 怎么去做上面的签名呢、就是用 fingerprint、这样就不用输密码了
15. Bootstrapping
    - 在重置密码的时候、如何通过邮箱里的 url 来重置你的密码呢？会涉及到一个参数。但是其中包含一个随机数（只有服务器知道）、所以一般人构造不出这个能够让你重置密码的 url（进而修改任何人的密码）
    - 对于随机数、很多时候通过设备里的/dev/random 来获取这个随机数（其中包含很多内容、比如键盘敲击的次数等）

- 一个随机性更强的方法就是用熔岩灯的照片

16. 下面讲的是一些 Security Principles
17. Principle of Least Privilege
    - 比如 linux 不要用 root、要用普通用户、这样可以保护自己不做 rm -rf /这种事情
18. Least Trust 最小化可信
    - Privileged components are "trusted"，要尽可能减少这类组件
    - 增加 Untrusted components （does not matter if they break）
    - trust 本身是好的、但是要尽可能减少
19. Users Make Mistakes
20. Cost of Security（安全是有代价的）
21. 下面开始讲 ROP 和 CFI
22. ROP （Return Oriented Programming）
    - 一开始讲了一个例子、是如何通过 buffer overflow 中注入一点点代码就执行很多操作的、就是只需要注入 batch 命令的对应的二进制代码就可以了（然后让一个机器变成一个 bash）
    - 所以有了 Defense: DEP (Data Execution Prevention)：就是用硬件来保证栈和堆可写不可执行、来避免上面的问题
    - 有了不可执行之后、第二个攻击是 Attack: Code Reuse (instead of inject) Attack。这个核心就是复用当前的代码片段（ROP）、所以可以在覆盖的时候、多覆盖几个地址、然后通过 return 将很多个代码片段粘在一起（后面的几张 ppt 讲的是攻击的方法）
23. 怎么去防御呢：第一种方法是藏起来所有的二进制文件；第二种方法是 ASLR、就是随机化地址空间、就是你看得到对应的代码、但是这样就不容易找到对应的地址了；第三种方法就是 canary、就是在栈上面放一个随机数（因为 canary 肯定放在了 return addr 前面）、然后在 return 的时候检查这个随机数是不是对的、如果不对就直接退出、去监测 overflow
24. ASLR
    - 每次运行程序的时候、二进制的地址都是随机的
    - 但是如果是 fork 的话、子进程的地址是和父进程一样的、就没法这样随机了（如果要保证还是随机的话、那么成本就是很高的）
25. CFI：Control Flow Integrity
    - 就是用来 defend ROP 的
    - ROP 的本质是覆盖掉原来的正常的 control flow、然后用攻击者自己的 control flow
    - 所以本质上需要在每次 return 的时候、返回 call 的下一行、而不是其他地方
    - 二进制语义中的控制流语义约束就没了
    - 所以需要在二进制代码中、把固定的控制流约束给它嵌入进去、就能控制了
26. 控制流主要有直接跳转和间接跳转
    - 直接跳转：地址是固定的（direct call，jump）
    - 间接跳转：地址是不固定的（return，indirect call，jump）
    - 这里有一个表、显示了大量的运行时的控制流都是间接跳转的（然而在二进制文件中、更多的是直接跳转的代码）然后下面的数据显示、程序跳转到的位置并不是那么地发散
    - has 1 target: 94.7%
    - <= 2 targets: 99.3%
    - > 10 targets: 0.1%

    - 所以能不能在 binary 里面加入程序的控制流的信息、让他不要乱跳
27. 这个上面有一个 CFG 的例子（Control-Flow Graph）
    - 具体是两个地方 call 同一个地方的函数、以及同一个地方 call、两个不同的函数
    - 12345678 是随机数、可以改的、用来比较是否是正确的 control flow，如果不一样、就跳到一个 error label（所以攻击者只能修改 ecx、而不能修改这个硬编码的值 12345678），因为其他地方也会检查
    - 这个硬编码的例子也有问题、因为有些是一些代码是发布的一个库、所以它不兼容（就是 library 里的代码已经是安全的了、但是实际应用程序里的代码可能还没有打 patch、所以就会导致出错）
    - 所以这里有一个工程上的小 trick 加入了一条 prefetchnta 12345678 的指令、它本身是一条可以执行的代码、但是这个 12345678 显然不是一个合法地址、所以它这个 prefetchnta 就偷偷地不执行了、就可以正常地运行没有打 patch 的代码了（可以正常执行那个硬编码的数字）（好处就是兼容性很强）
28. CFI 还是有问题的：Suppose a call from A goes to C, and a call from B goes to either C, or D (when can this happen?)
    - 这个会带来一个问题、就是原来 A 调用 D 是非法的、但是 A 和 D 的 label 是同样的 12345678，就导致了 A 就能调用 D
    - 还有一个问题、就是这个关于 A、B、F 中，return 的标签都是一样的（就是两个地方调用一个地方的问题）（比如说 A 到 F，B 到 F，然后 F 返回的时候、可能会返回到 A 或者 B，这个时候就会有问题）
    - Solution: shadow call stack
    - Maintain another stack, just for return address
    - Intel CET to the rescue (not available yet)
    - Shadow Call Stack 的工作原理
    - 维护一个额外的栈：程序在原始栈之外，维护一个独立的 影子栈，只用于存储函数调用时的返回地址。这个栈与程序的主栈相对独立，专门用来存储每个函数调用的返回地址。
    - 分离保护：这样，如果栈上的数据被恶意修改，攻击者可能无法影响影子栈中的返回地址，因为它们是独立的。即使栈本身的内容发生变化，影子栈中的返回地址仍然可以被验证。
    - 保护验证：当函数返回时，程序不仅要检查主栈上的返回地址，还要检查影子栈上的返回地址。如果两者不一致，程序就会检测到攻击，并可以采取适当的防御措施（例如终止程序或抛出异常）。
    - 优势：防止栈溢出攻击：通过分离和保护返回地址，防止攻击者通过栈溢出修改返回地址。
    - 劣势：提高了内存和性能开销：由于需要维护额外的栈
29. 所以需要做一个优化 CFI: Control Flow Enforcement
    - 一个优化保证就是要就是做到一对一
    - 每个标签都需要是唯一的，但是很难做到（需要复制很多的 call、如果一个 call 的地址有 10 份）
    - 所以目前的选择是目前的 CFI 并不会这么细、目前还是粗粒度的 CFI
    - 这个是微软提供出来的（CFI）
30. CFI: Security Guarantees
    - 防御了 Stack-based buffer overflow, return-to-libc exploits, pointer subterfuge

    - 但是不防御 Does not protect against attacks that do not violate the program's original CFG （就是不防御不违背控制流图的攻击）
31. 后面这个 Possible Execution of Memory 的图就显示了在不同的实现下面、程序会有不同的可能的执行内存地址
    - 首先是加不能执行的栈
    - 然后再加入 RISC、只能跳到对齐的头
    - 然后再加入 CFI、就是只能跳到对应的地址（控制流图加入）

28 Control Flow Integrity & Secure Data Flow

1. 上面主要攻击的方式还是 buffer overflow。
2. 防御方式：目标地址前面的随机数要是一样的，不一样就直接退出。加大攻击者的难度，但是不是完全消除危险。（就像加盐、只是增加了攻击者的代价）
3. 还有的方法：就是随机化地址、因为不同机器上的内存偏移不一样、所以就不容易找到对应的地址了。（从工程的角度，提升攻击难度）
4. 讲了一个具体的攻击例子 Blind ROP （defeat ASLR）
5. 它貌似就是通过发送 http 请求的方式、将这个请求变成一个 bash、然后就可以执行一些命令，然后控制这台机器了
6. 这个能够知道的假设：就是攻击者唯一知道机器里有一个 buffer overflow 的 vulnerability(能通过很多种方式知道)，但是不知道这个机器的地址空间是在什么地方、也并不知道它的二进制。
7. 它有可能 Get 一个内容、然后返回一个 404（正常返回），就是找不到。也可能 get 一个可能造成 buffer overflow 的内容，然后会跑飞、就是 connection closed（立刻关掉）。还有一种情况就是不返回任何消息(hang 住了)。（一共三种反馈）
8. 第一种方式是 stack reading，就是通过不断尝试覆盖掉多大的数组大小、当什么时候报错、就能正好找到对应的 return 的地址（正好一个 byte，碰到 return address）然后一直试，直到变到不崩、就找到了正确的 return address 的位置。（可能会试几百次）（为什么有地址随机化、还是一样呢？因为 nginx 里面服务某一个的进程是 fork 出来的、由于父进程和子进程的地址是一样的、所以就不会有地址随机化的问题）（只有父进程 crash 了、才是随机化）（如果不是这样的话、有 100 个子进程但是挂了 1 个、会需要把所有其他的子进程都重新启动，不科学）
9. 怎么找到能够为我所用的代码片段 gadgets 呢？首先需要让服务器把它的二进制传过来。这样就需要先执行 gadgets 运行一些代码，所以很难。
    - stop 就是 hang 住了
    - crash 就是 connection close 了
10. 这里就是讲了尝试跳转到不同的内存地址、然后找 gadget 的一个过程。这里会有三种不同的 gadget，stop gadget（从不 crash），crash gadget（总是 crash）和 useful gadget（会有 return）。
11. 这里有两种 crash 的原因、第一种是 401170 本身就会 crash、还有一种是因为 401170 return 之后、执行了 other 的 gadget、然后导致了 crash、这里可以通过修改 other 的地址、来判断这个 401170 是不是第三类 gadget。（如果修改之后会 hang 或者会直接 crash、就代表这个修改是有效的）
12. 找到了 useful gadget 之后、就要找 write system call。这个 sock 就是传给攻击者的 socket、buf 就是对应要写的地址内容、比如 401170，然后这个 len 也是越长越好。现在就是要让 buffer overflow 的部分地址去跳转到这个函数部分中。这个 rdi、rsi、rdx 就是传参的寄存器，然后再 call。为什么是 pop rdi；ret 呢、因为攻击者是控制了栈上的内容、所以就相当于把栈上的内容 pop 到 rdi 里面去。

13. 在所有的 x86-64 的函数代码中、都会有一段 pop 寄存器的代码、因为这些个寄存器是 callee-saved 的、所以在函数调用的时候、会把这些个寄存器的值保存起来、然后在函数结束的时候、再把这些个寄存器的值 pop 出来。
14. 这个 BROP（Blind rop）gadget 就是通过不断地修改这个栈上的地址、如果中间夹了 6 个 crash 的地址、然后最后修改上面的地址是 crash 或者 stop（hang），就可以确定这个 gadget 是不是把中间六个地址都给 pop 出来了、就可以获取这个 gadget（这个 gadget 可以解决 rsi 和 rdi 的问题）
15. 但是还是不知道怎么解决 pop rdx，因为 rdx 是 caller 的，所以就把这个变成了 call strcmp（因为可以 call write、所以这个 strcmp 就在这个 write 边上，就相差 8byte，因为这个是 libc 的函数）因为在运行 strcmp 的时候、rdx 是一个 strcmp 的参数、所以 rdx 就会被设置成这个 string 的长度，后面穿的这个参数的长度就会变成 rdx（具体是多少、取决于这个 传进去的 string 的指针对应的字符串的长度是多少）（其他修改 rdx 的函数也可以）
16. 最后是要怎么找到 write 的地址。通过 PLT 表、找到动态链接，然后会有 GOT 表、就把 libc 里的 write 的地址给找到了。
17. 对于攻击者、找到一个地址之后、由于是 8byte 对齐的、如果跳 16byte 之后、都是 useful gadget、那就找到了 PLT 表、这个表有规律性、就有很多的 gadget。
18. 另外，大部分的 PLT 项都不会因为传进来的参数的原因 crash，因为它们很多都是系统调用，都会对参数进行检查，如果有错误会返回 EFAULT 而已，并不会造成进程 crash。所以攻击者可以通过下面这个方法找到 PLT：如果攻击者发现好多条连续的 16 个字节对齐的地址都不会造成进程 crash，而且这些地址加 6 得到的地址也不会造成进程 crash，那么很有可能这就是某个 PLT 对应的项了。
19. 有了 PLT 表、怎么找 strcmp 呢？可以设置参数的特征、然后一个个地去尝试、满足这个 crash 表的特征的就可能是 strcmp（控制参数模式）
20. 最后是找 write。通过 socket 能不能收到 server 的信息、来判断是不是 write sys call（如果正好跳到的是 write、就会在收到的消息里面有很多乱七八糟的值）
    - 然后就可以通过 write 的方法、把整个服务器的二进制 write 到 socket 里面
21. 用同样的方法、可以通过 PLT 表找到更多的 gadget、比如说 execve、然后就可以执行 shell 了
22. 最终的次数里、stack reading find PLT 和 find BROP gadget 要的次数最多的。
23. 前面讲的都是 control flow、攻击者劫持执行的一个角度，也不能从根本上解决问题、因为有可能代码本身写的就是错的、所以没法解决这个问题，所以后面讲了 data flow protection（初心是为了要保证安全，比如说数据不要被人偷走）
24. 有两大类追踪数据的方法：方法 1：跟踪数据在程序中的流动。（正向的，保证这个数据不泄露）
25. 方法 2：（反向的）比如机器学习中的数据、要保证这个数据不会被修改（防止恶意数据，数据投毒）（可疑数据需要隔离开来，检测可疑数据的流动，然后可以删掉被污染的模型）
26. 比如说有手机上的 key logger、记录下你的按键的位置。或者通过钓鱼 app、记录下密码。
27. 还有就是直接扫物理内存，比如说某些邮箱软件、在打开之后会把邮件内容和密码直接保存在内存中、所以直接扫一遍内存就能偷到大量的密码。
28. 以及截屏的权限、也是会有能够获取密码的风险。或者低温让手机冻住、然后内存丢失的速度就会慢，就能找到所有的文件了。以及手机的倾斜。
29. Tainting: Data Flow Tracking
30. taint tracking：敏感信息的生命周期一定要最小化。然后会有换页、所以内存里的数据会被不定期地换到硬盘里、而 OS 完全不知道、所以会有泄露的风险。
    - 跟踪来自外部的 input 数据

31. 有三个步骤：初始化、传播、检查。如果前面的不是常量、后续就会传播 taint、如果是变量、taint status 就会不断传播 true。检查就是通过 jmp 的方式或者 send，可以通过程序员去决定。所以 jmp 能不能运行、取决于这个值是不是 taint 的，是动态的。
    - 这个 taint、如果不加检查、会被覆盖、比如打压缩包，就会让 taint 丢失，所以有了下面的详细方法
32. taintDroid：关键是怎么做 propagation。首先，在 java 里面所有应用程序所有的数据、对应一个 taint 传递（内存级别，成本很高，taint 值对应也要操作）然后到了 IPC 这个级别、只要有数据的一个 byte 是被 taint 了、就认为这个数据是 taint 了，然后去跟踪跨应用的数据传递（比如说一个应用有 gps 权限、另一个没有、如果它们之间有 rpc、那么就不安全）然后最后在 untrusted app 的 sink 的时候、如果发现有 taint、那就可以卡住、就能保证安全。
33. 有一个 taint propagation 表。就对应了 taint 传播的规则（就是监测一个是否是外部传进来的函数指针、把对应本来的指针内容给覆盖了）
34. defending malicious input。怎么找 bug
    - 首先决定找什么东西的 bug、比如开源 library
    - 找到攻击攻击模块
    - 然后定位到攻击数据
    - 最后跟踪数据是怎么流的
35. 下面举的例子是 ffmpeg
    - 出 bug 最多的是 parser
    - 越是不常用的格式代码，bug 越多（4xm.c）
    - 它做了一个 unsigned int 赋值到 int 的一个语句
    - 然后在数组 tracks 里面做负数的索引，会导致内存写的问题、然后就可以做任意内存任意写的操作。
36. 能不能通过 taint 来找到这个 bug 呢、可以、因为是一个用户传进来的值。分为 seed、track 和 assert。但是 TaintCheck 方法的问题是性能很差。overhead 非常高
37. 最后这个图讲的就是、如果一个应用是 io intensive 的、而不是 cpu intensive 的、就适合用 taint 方法，否则会造成很多性能损失。

29 Privacy & Review

1. 设备和远端的设备通信的时候、就需要进行很多加密了
    - 第一个操作是加密，这个是为了保证数据的机密性、但是不保证数据的完整性
    - 第二个操作是算 MAC，就是不知道这个 key、就算不出对应的 token，可以保证数据的完整性（如果被改了、会对不上）（是一个加密的 hash、可以把 token 和 message 放在一起传播）（就是只有你能够发布这个 message，别人做不了这个事情）
2. 但是上面的 MAC 和加密、并不能防止 replay attack，就是有一个中间人拿到了对应的 c|h、然后重发之后、还是能打开对应的锁（因为密文本身是合法的）所以可以加上一个 sequence number、是不断往上加的、这样就能防止 replay attack（bob 会忽略掉对应的 c|h）
3. 但是就算有了 seq number ，还是可以做 reflection attacks（就是 alice 和 bob 发的消息是对称的，因为不同的几个消息可能 sequence number 是一样的，所以攻击者可以通过反射的方式，把消息发回给 alice、但是正真要发的消息就发不过去了。）
4. 所以就要让消息变得不对称，可以通过两个密钥的方式、比如 alice 用密钥 a 加密、用密钥 b 解密、bob 用密钥 b 加密、用密钥 a 解密。这样就能防止 reflection attacks。（不对称）

5. Diffie-Hellman Key Exchange，就是使用了一个 mod 和次方的数学性质、然后让两端可以用同一个 seed 去做加密，最终得到了一个统一的 key
6. 但是上面的方法还是没法避免 Man-in-the-Middle 的攻击、因为没法验证对方是不是确实要通信的人
7. 所以有了 RSA 的算法，分为了公钥和私钥、这里的场景是 bob 用自己的私钥加密、然后把公钥发给 alice、然后让 alice 用公钥解密、能解出来就代表是 bob 发的消息。还有一个场景是如果不知道 bob 在哪、alice 可以广播一个用 bob 的公钥加密的消息、然后 bob 就能用自己的私钥解密、然后就能知道是 alice 发的消息了。
8. 公钥可以从机构获得，可以知道 bob 的公钥是多少
9. 所以解决方法是、当我访问支付宝的时候、支付宝会把自己的公钥加上域名加上第三方的签名发给我（合起来就是证书）、然后我就能用这个公钥去解密、然后就能知道是支付宝发的消息了。
    - 然后可以把这个公钥加上签名去机构算一下、如果能算出来就代表是真的
10. 如何保证这个机构的公钥是对的呢，是在这个浏览器里自带的，二十几个
11. 量子计算能够直接从证书反推产生这个签名的明文、所以就破坏了 RSA 的安全性
12. privacy 使用技术的手段进行保护
13. OT：Oblivious Transfer
    - 一个例子就是 alice 有两个文件、bob 要获取其中一个文件、但是 alice 不知道 bob 选的是哪个文件
    - 但是不能让 bob 拿到所有两个文件，只能给它一个
    - 这个是通过 OT 协议来实现的
14. 1-out-of-2 OT
    - 就是 bob 只能最后得到两个密钥中的一个，其中解密出来的 k 就是对应的随机数 r（这个 k0 和 k1 之间只有一个数是 r）、只有 bob 知道
    - bob 最后再通过做异或操作的方式、就能得到最后的加密信息
15. differential privacy
    - 一个例子就是 bob 有一个数据库、然后 alice 有一个查询、然后 alice 通过查询这个数据库、就能知道 bob 的信息
    - 但是 bob 不想让 alice 知道自己的每一个条目的信息
    - 可以通过限制 query 的次数、或者让返回值不精确、或者是固定规则、不让它返回对应的单独人的工资条目、所以是有损的。
16. Secret Sharing
    - 比如有 10 个密钥、要同时有 6 个人同时在才能打开这个锁
    - 好处：每个人节省空间（比如说可以只存 70%的数据），然后节省了 communication cost
    - 缺点：communication overhead 会很高
17. Homomorphic Encryption
    - 就是希望搜索 但是不希望看到明文
    - 可以基于密文做索引，就是 google cloud 可以不知道你的搜索内容，但是可以给搜索做索引
18. SWHE: SomeWhat Homomorphic Encryption
    - 有一个同态加和同态乘的操作
    - 就是密文相加和相乘 解密之后就是对应的明文相加和相乘
19. Trusted Execution Environment (TEE)

    - 云服务上的加密内存、加密虚拟机的服务
    - 内存是密文、可以从内存到 cache 的时候解密
    - cache 里面是明文
    - 信任方变成了处理器厂商
    - 特点 1：Isolated execution 运行时隔离、保证获取不到对应的内存数据
    - 特点 2：Remote attestation 远程验证
20. Process of fixing a bug/vulnerability
21. Phase-1: finding a bug
    - Survey whether the bug has been found
    - Simply the process of re-producing bug 复现
    - Evaluate the seriousness, if belongs to security, offer an exploit
    - Provide a patch if you have one.
22. Phase-2: Report
    - Not security bug：kernel mailing list, bugzilla
    - Security bug：security@kernel.org
23. Phase-3: handle the bug
    - Related developers and maintainers join the discussion
    - Patch proposing and discussion
    - Test and patch review
24. Phase-3.5：Embargo Period （可选）
    - discuss with maintainers of distributions, prepare to fix
    - Why not just fix it?
    - Submit a patch in public mailing list means public
    - Prevent distributions not fix the bugs in time
    - Decide whether embargo period is needed: evaluate
25. Phase-4: Public
    - Merge to mainstream
    - Backport to related stable versions
    - Publish to public mailing list, security group
    - E.g., oss-security@lists.openwall.com
    - Update CVE status to public

30 Review

1. ROP 的本质：控制流劫持、以预期的方式去执行代码片段
2. canary 可以通过 BROP 被干掉、就是一点一点读出来
3. 这个 taint 也可以通过一些方法把 taint 洗掉、比如用 if else 来判断（a==1 时、b=1，a==2 时，b=2）、然后就可以把 taint 的值给覆盖掉
4. 曼彻斯特编码、为了防止这个连续的 1 和 0 出现、导致读不到时钟信号的变化、因此加入了变化
5. dns 能够 scale 的本质、去中心化的管理、找到最近的服务器节点去加就可以了

6. raft 解决了 replicated state machine 的问题
    - 大部分时间 leader 在发 rpc、follower 在接 rpc
    - 每一个任期内只能有一个 leader
    - term 号最高的作为 leader
    - heartbeat 不能随便设、不然选不出来、一直重试
    - 超过 majority 的节点认为成功了、就 commit（leader 先）
    - 维护的是前缀的 consistency，那么如果一个 entry 是一样的、那么前面都是一样的
    - 为了实现 prefix consistency，appendEntries 会检查前一个 entry 是不是一样的，如果不一样，就会返回失败
    - 如果 commit 了、可以 overwrite
    - 额外的机制，哪些 entry 是 commited 了
    - 还有一个是一个 entry 不会被 overwrite 掉
    - 只有当前最新的机器才有资格做一个 leader（看 term 是否是更大的、如果一样、那就看 log 是否是最长的）
    - snapshot 也要复习一下
7. master 的 failure 会比较罕见