【计算机系统工程】网络

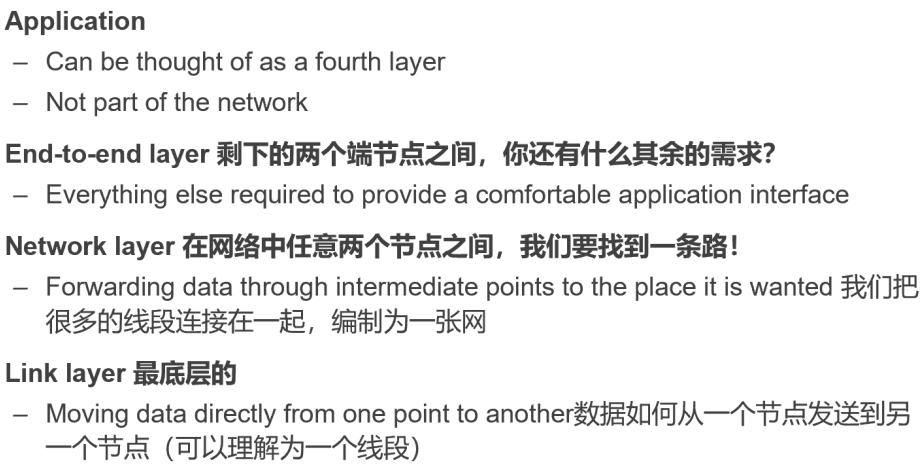
@credits Yubin Xia, IPADS

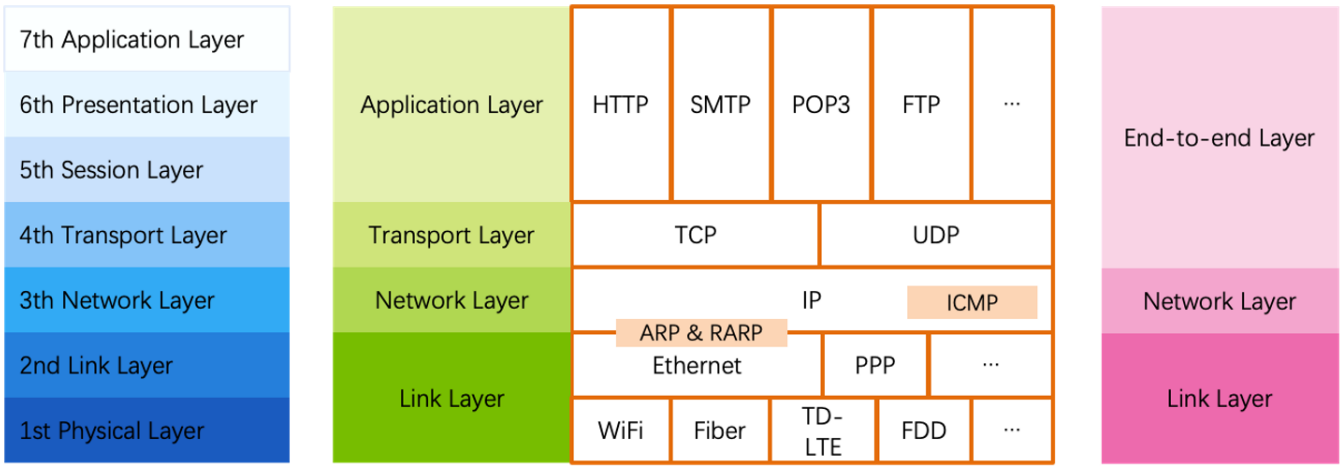
1. Network-brief intro

1. Layering

网络是一个很好的全是控制系统复杂性中的layer的方式。复杂系统因为组件之间的交互太多了，所以很难去精细地控制。所以我们提出了MAHL四个方法：

1. Modularity
2. Abstraction
3. Hierarchy
4. Layering





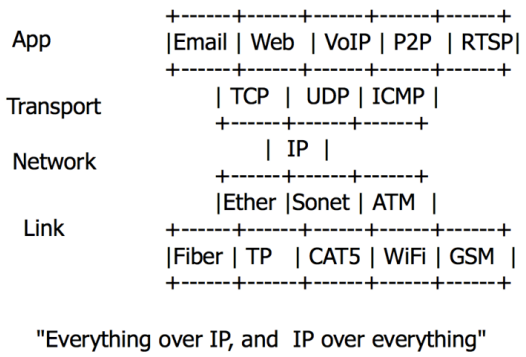
1. End-to-end Layer：强调的是从一个点怎么找到另一个点，一个例子就是发快递，只需要填自己的地址和对方的地址；我不care中间的具体的过程：比如实际上可能从快递站->中转站->...->家。**【在我们的课程中，我们认为Application layer和Transport Layer都是端到端的协议】**
2. Link-layer：一段一段怎么走？可能是骑着电动车的快递小哥，也可能是开着卡车的送货员。所以link layer在网络里有很多方法，比如WiFi、有线以太网、有线电缆、光纤……
3. Network Layer：就是把Link Layer一段段连起来，找到从一个点到另一个点的最短路径，但是特殊的点就是找的时候那条路径可能还在，但是在发包的时候可能就没了。所以Network Layer只负责找到一条路，但是这条路通不通并不保证。End-to-end Layer如果发现发包的时候这条路是断的，那么我们就让Network Layer再找一条路。

我们发现，在底下有以太网，已经有一点点偏传输了，最下面还有Fiber、光纤等。在这样一个Protocol下，它是一个典型的沙漏结构。

More people, more useful
- Value to me = N
- Value to society is N^2

Network, dumb vs. smart
- Standardize vs. flexibility

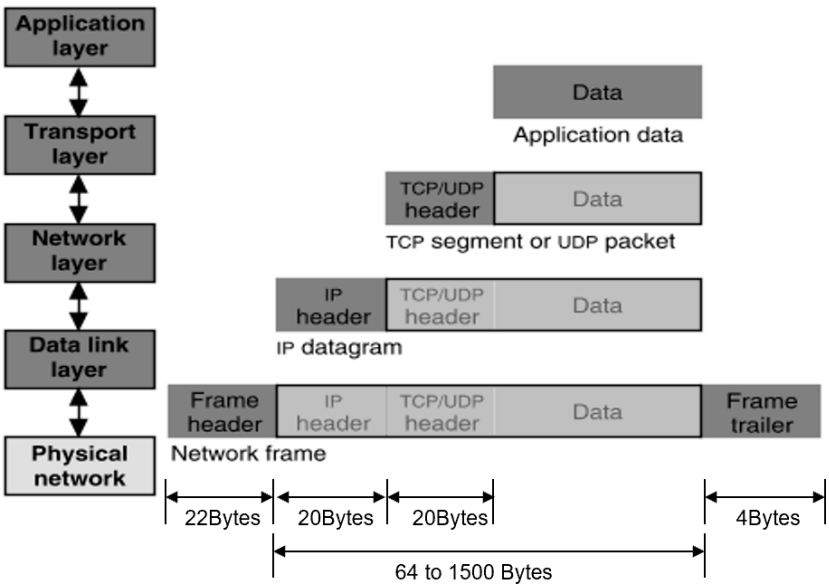
Network is a black box
- Simplify the system that uses it



> Q：中间是一个很窄的点，为什么是一个沙漏型呢，也就是为什么中间只有一个IP呢？
>
> A：因为网络的价值取决于加入网络节点的数量。它期望的是越多的机器连在一起越好。
>
> 一旦有了IP之后，它就成为了一个占据了最大优势的协议，把所有的节点连接在了一起，这样我们上层就不用care下层到底是怎么连的。
>
> 相似的场景我们可以推广到OS，OS的下层是千奇百怪的硬件，上层是各种各样的软件，而OS就那么几个，成为了一个沙漏的中间部分。这样应用程序就不需要做很多兼容。所以沙漏模型是和生态紧

> 密相关的，现在所说的卡脖子，其实就是卡中间这个瓶颈。比如哪天IP协议不让用了，就牵扯范围非常大。

2. Package Encapsulation 包的封装



整个就像一个信封一样，信纸外面一层一层包着信封。

发到transaction layer，就会加上TCP/UDP的包头，到了Network Layer就会加上IP的包头，到了Data Link Layer就再加上了以太网的包头，这也是我们为什么使用stack来表示网络协议，因为它就像一个stack一样。

> Q：为什么我们把数据切分成64~1500 Byte的小的包的形式去发送，而不是以流数据的形式一直发送呢？
>
> A：这也是一个trade-off。我们包小了，利用率就低了，overhead就变高了。如果我们不切包，整个一起发，那么一旦错了一点点就要整个文件重发，效率低。而如果我们切了太细了，那么导致包头的overhead又比较大。所以64~1500 Byte也是一个经验的数据。

3. Brief intro of each layer

1. Application Layer

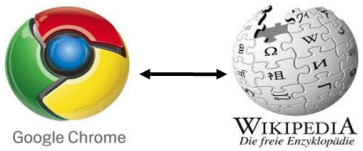
Application Layer

Entities
- Client and server
- End-to-end connection

Name space: URL

Protocols
- HTTP, FTP, POP3, SMTP, etc.

What to care?
- Content of the data: video, text, …




```
<html>
  <head>
    <title>Google</title>
    <script>window.google=…..
    </script>
  </head>
  <body>  …  </body>
</html>
```

2. Transport Layer

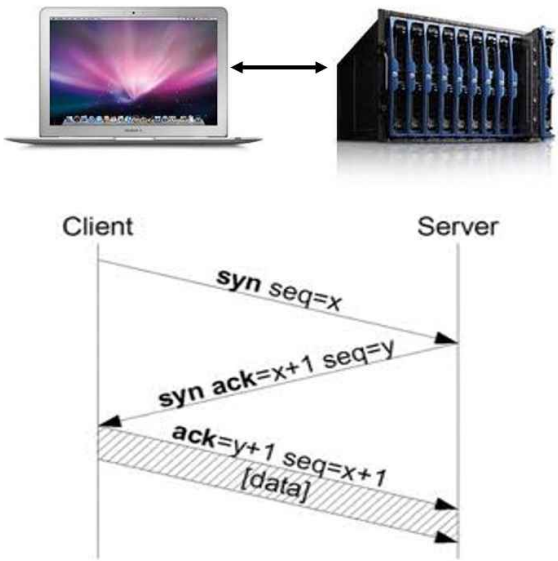
Transport Layer

Entities
- Sender and receiver
- Proxy, firewall, etc.
- End-to-end connection

Name space: port number

Protocols: TCP, UDP, etc.

What to care?
- TCP: Retransmit packet if lost
- UDP: nothing



Transport layer需要考虑去识别不同的session，比如我们的服务器既服务了同学A和同学B，我们在响应的时候怎么保证同学A和B收到了自己所对应的包。那么我们就要加一个namespace，也就是port。当我们连到一台服务器的时候，大家通过port和application连接，所以在一台笔记本上，我们就可以同时发起65535个请求。因为一台机器只有1个IP，我们就可以使用port来做扩展。

Packet Format of TCP & UDP

TCP Segment Header Format

Bit #	0 – 7	8 – 15	16 – 23	24 – 31
0	Source Port		Destination Port	
32	Sequence Number			
64	Acknowledgment Number			
96	Data Offset	Res	Flags	Window Size
128	Header and Data Checksum		Urgent Pointer	
160...	Options			

UDP Datagram Header Format

Bit #	0 – 7	8 – 15	16 – 23	24 – 31
0	Source Port		Destination Port	
32	Length		Header and Data Checksum	

也就是传过去的时候，client会给server一个sequence，也就是告诉server：“接下来会传输给你一个包，这个包的编号就是x”，Server收到以后会返回ack=x+1，并且把自己的一个y作为seq返回给client。下一次client发送的时候就继续发送ack=y+1,seq=x+1和自己想发送的数据。

如果一共发送了1000个包，那么seq就会被加1000。这样的好处就是，我们作为旁边一个人看到两个人在发消息，我们想冒充其中一个人来发消息，但是我们猜不到sequence number是多少，这就很难冒充。所以sequence number起到了避免旁边人冒充的功能。

3. Network Layer

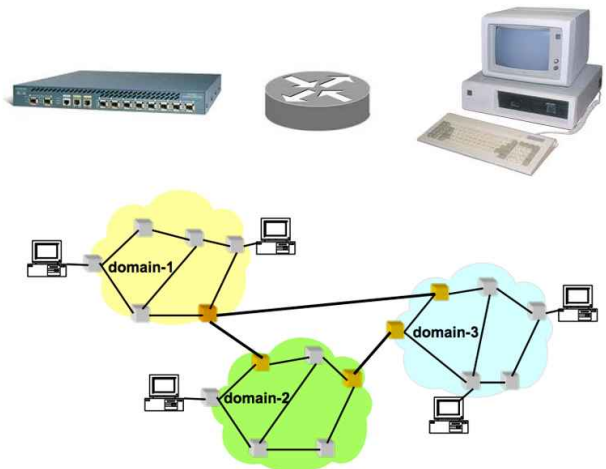
Network Layer (the Internet Layer, IP Layer)

Network entities
- Gateway, bridge
- Router, etc.

Name space
- IP address

Protocols
- IP, ICMP (ping)

What to care?
- Next hop decided by route table

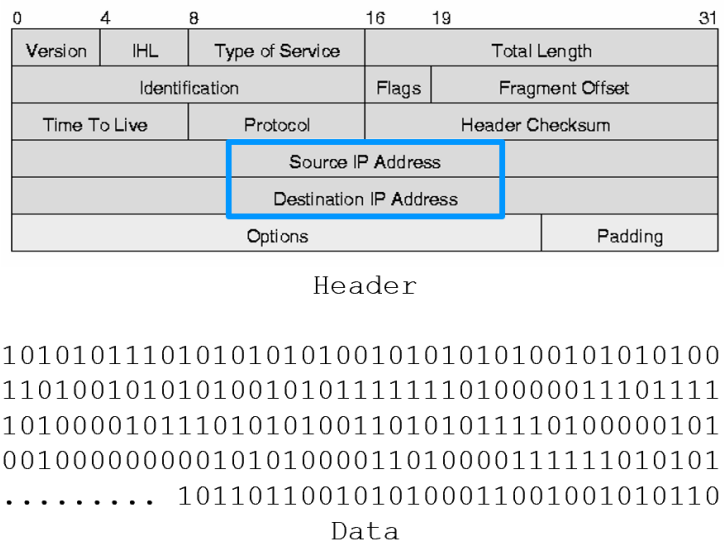


到了Network Layer后，我们看到了就一台台这样的设备了。我们要考虑怎么去做路由，怎么去找线段。IP这个Layer中，ICMP可以认为是在transport layer和network layer的中间。

Network Layer需要关注的是网关、网桥和路由器，整个Internet就是建立在很多个路由器互联的基础上的。那么我们该怎么知道谁在哪、我要发给谁、www.baidu.com的IP在什么地方。我们就需要用到路由表，这些IP设备最核心的功能就是维护路由表。

> 路由表是一个非常典型的分布式系统，因为没有一台机器可以保存整个世界的网络的**最新的拓扑结构**，每个人手里只有一小块地图，那么我们是如何通过每个人手里的地图碎片，把信息从交大发到哈佛的呢？

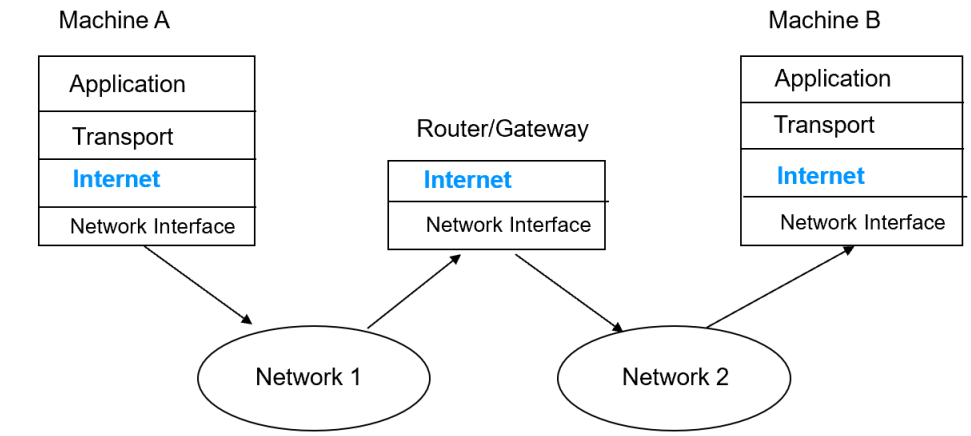
IP Datagram (Packet, Package)



包核心的部分就是source ip address和destination ip address。一个网卡上可以设置很多个虚拟IP。所以IP这个概念就是一个逻辑上的概念，并不会直接地和某个设备绑定。

Time to live是什么呢？它的意思是做多跳几跳，每次从一个节点到另一个节点就-1。减到0的话这个包就被扔掉了；否则网上就会存在很多流浪的游魂package。

4. TCP/IP 架构



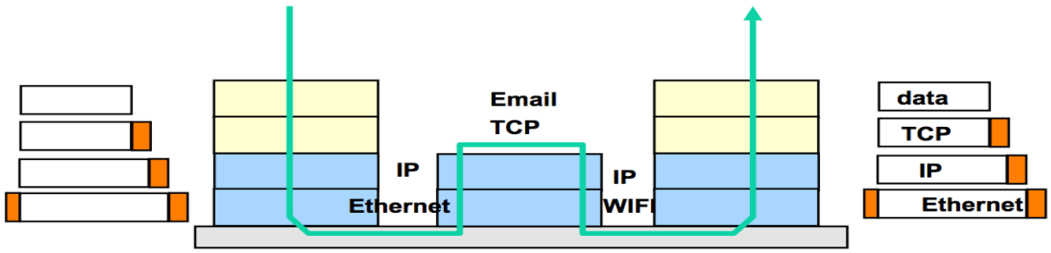
TCP/IP Architecture

Each layer adds/strips off its own header

Each layer may split up higher-level data

Each layer multiplexes multiple higher layers

Each layer is (mostly) transparent to higher layers



- transparent：透明的

2. Network-link layer: from a node to its physical neighbor

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
- 3. Framing bits & bit sequences
- 4. Detecting transmission errors
- 5. Providing a useful interface to the up layer

目标就是从一个点到另一个点传输数据：

1. 在物理上，就是最底层地传输数据
2. 要考虑如何让一条线被很多人复用？
3. 考虑如何把数据变成实实在在的编码传输信号？
4. 找到传输过程中的错误，避免重传
5. 对上提供interface

一些概念，不详细展开：曼彻斯特编码、海明距离。

3. Network-network layer: all about routing

当我们能在两个节点之间传输数据了之后，那么我们怎么用这一段段数据来拼成一个网络层呢？这就是我们常用的IP协议，它是best-effort network，如果找不到就丢掉。如果我们在网络这一层来做absolute guarantee就很复杂，底层还是best-effort即可。

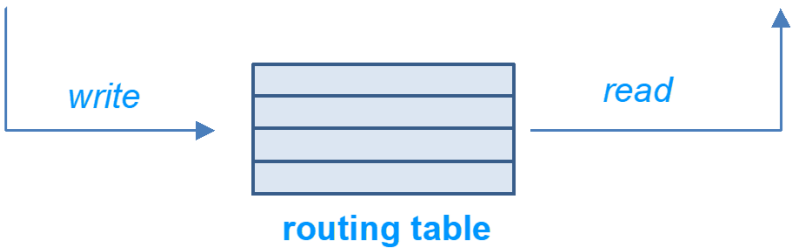
对于网络层来说，它对外有一些attachment point（AP），当我们接到网络的时候，我们要找到AP，把设备通过AP（WiFi、网口）接入网络。AP对应了一个数，就是一个网络地址，一旦知道了网络地址，数据就通过AP来发送传输。主要是发包接口和收包接口（network_handle）。在网络中，我们不仅仅是一个终端节点，还有可能是中间节点去做一个forward，这就是我们说的路由器，对于路由器来说，大量的包只是中转一下。

那么既然说到中间有一个中转的过程，routing就是中转的过程。既然我们有一个目标和起始地址，网络层怎么找到这条路径呢？

其实一个机器只要插了两个网卡（一个内网、一个外网）就可以做一个简单的路由器。路由器通常有很多个网口。它的本质就是插着很多网卡的电脑。高端路由器非常贵，因为在处理的时候对吞吐量要求非常高。

整个路由器是怎么工作的呢？其实非常简单。每一个router都有多个网口，它核心的数据结构就是一个routing table，记录每一个口对应了哪一个网络。有人认为这上面都不需要跑OS，只需要跑网卡驱动和数据结构即可。不过有了OS之后，我们可以做防火墙、流量管控等。

在这里我们就要讲**控制面和数据面的分离**，这是一种经典的分离方式。



1. Routing 控制面：如何生成路由表？

总体来说routing的协议就是让每个交换的节点知道该怎么走能到destination，希望尽可能找一条cost最小的路径。注意，这**并不是距离最小的路径**。

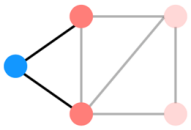
> Q：怎么访问github最快呢?
>
> A：最快的路径可能是我们先到阿里云，再通过专用通道到美国的阿里云去访问github，这可能在距离上多一点，但是速度会更快。

所以这并不是完全依赖距离。然后，完成之后我们就要构建出routing table。注意，每时每刻都有不同的服务器crash、网络的拓扑结构在不断变化，所以路由表也要变化。

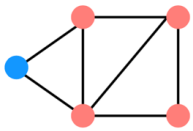
1. 构建分布式路由：从地图碎片到整个世界

Distributed Routing: 3 Steps in General

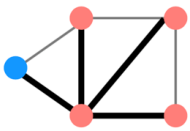
1. **Nodes learn about their neighbors via the HELLO protocol**



2. **Nodes learn about other reachable nodes via advertisements**



3. **Nodes determine the minimum-cost routes (of the routes they know about)**



Two Types of Routing Protocol

Protocol 1: Link-state

- A node's advertisements contain a list of its neighbors and its link costs to those nodes
- Nodes advertise to everyone their costs to their neighbors
  - via *flooding*
- Integrate using **Dijkstra's shortest path algorithm**

Protocol 2: Distance-vector

- Nodes advertise to neighbors with their cost to all known nodes
- Update routes via **Bellman-Ford** integration

换一种思路来理解，就是第一种给所有人发，然后呢这个广告是比较小的；第二种是给自己的邻居发，但是广告是比较大的。

2. Routing 数据面：查表+转发

3. Scale Routing

3 Ways to Scale

Path-vector Routing

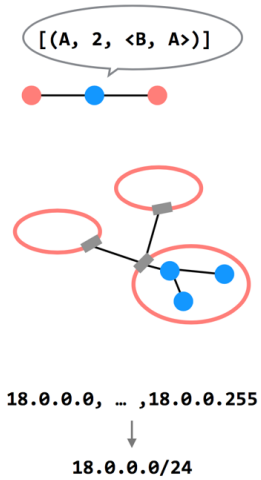
– Advertisements include the path, to better detect routing loops

Hierarchy of Routing

– Route between **regions**, and then within a region

Topological Addressing

– Assign addresses in contiguous blocks to make advertisements smaller

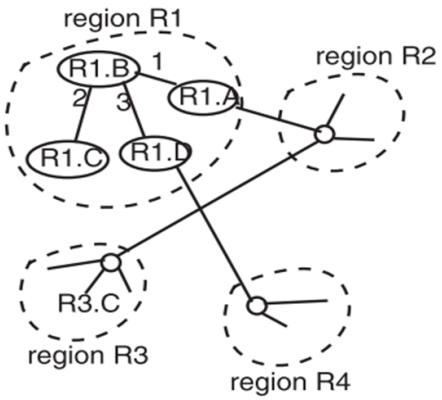


1. Path vector：我们把path也作为一个vector，advertisement中传的信息多一点，并不会花太多时间。也可以避免infinity问题。
2. Hierarchy of routing：我们不能要求每台机器上都记录全世界的所有主机的路径，我们可以让到美国的路由只有一条路。
3. Topological addressing：压缩连续地址，来做路由表的压缩。（这里就提出了子网掩码的概念）

1. Hierarchical Address Assignment

每一个节点都必须要有一个唯一的地址；节点数量越多，path vector就会越长；现在这个思路就是把一个节点的地址从原来单独描述这个节点变成region+station，两者各自有一个编号——你是XX号region里的yy号机器。

比如region11，number 75，这样我们就先去找到region 11，再去找其中的第75个机器。所以网络地址会被分成两部分，第一部分是region，第二部分才是station，eg：11.75。找的过程会有两次，而不是一次。因为整个region11只有1个entry，这样就大大降低了路由表的大小。



forwarding table in R1.B

region forwarding section

to	link
R1	local
R2	1
R3	1
R4	3

local forwarding section

to	link
R1.A	1
R1.B	end-layer
R1.C	2
R1.D	3

我们的路由表只需要记录4个region。

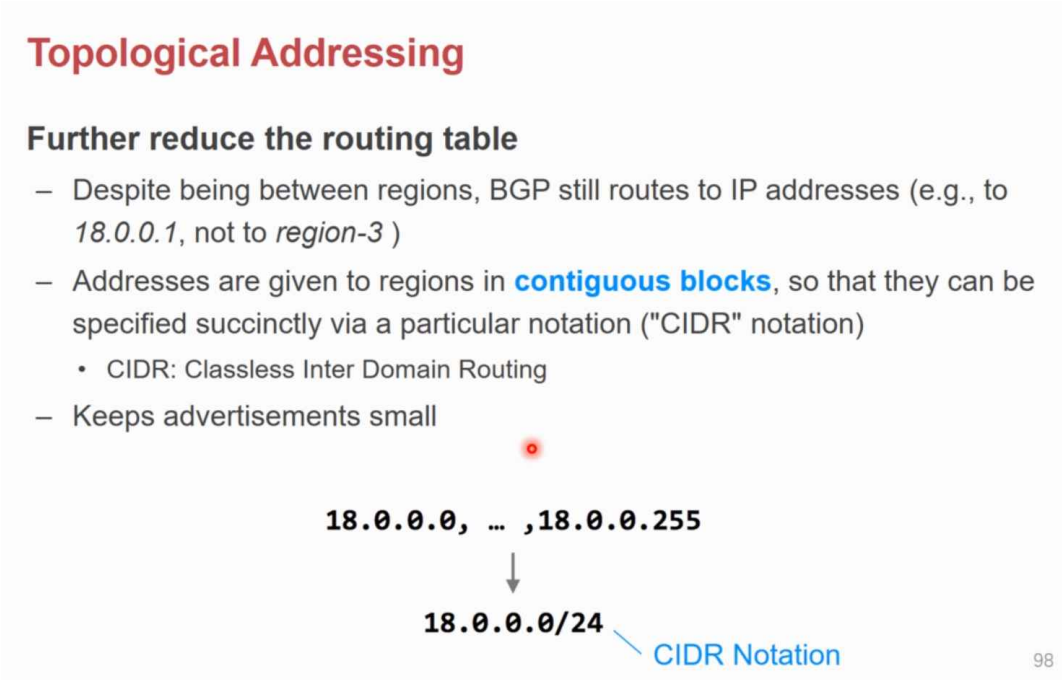
对于R1.B来说，只需要记录两个table，一个是region级别的，还有一个是内部的region内的路由表。

Hierarchy有一大类问题，因为它把address和location绑定在了一起。

我们的IP地址为什么要因为我们的地理位置发生变化呢？从最最直观的角度来说，物理地址和IP地址应该是没有关系的，但是我们为了降低路由表的大小，我们才选择把IP和地址绑定在一起。

2. Topology Addressing

1. https://zhuanlan.zhihu.com/p/371400090
2. https://blog.nnwk.net/article/152



这里的24就代表24个1。来一个ip地址，和开头连续24个1的二进制ip and一下，如果等于这个值（18.0.0.0），那么路由表就match。

这里的好处是显而易见的——路由表里面我们不需要每个ip单独去写，而是可以写成这样：

网口	路由表规则
1	18.0.0.0/24
2	18.1.0.0/22
3(default)	0.0.0.0/0

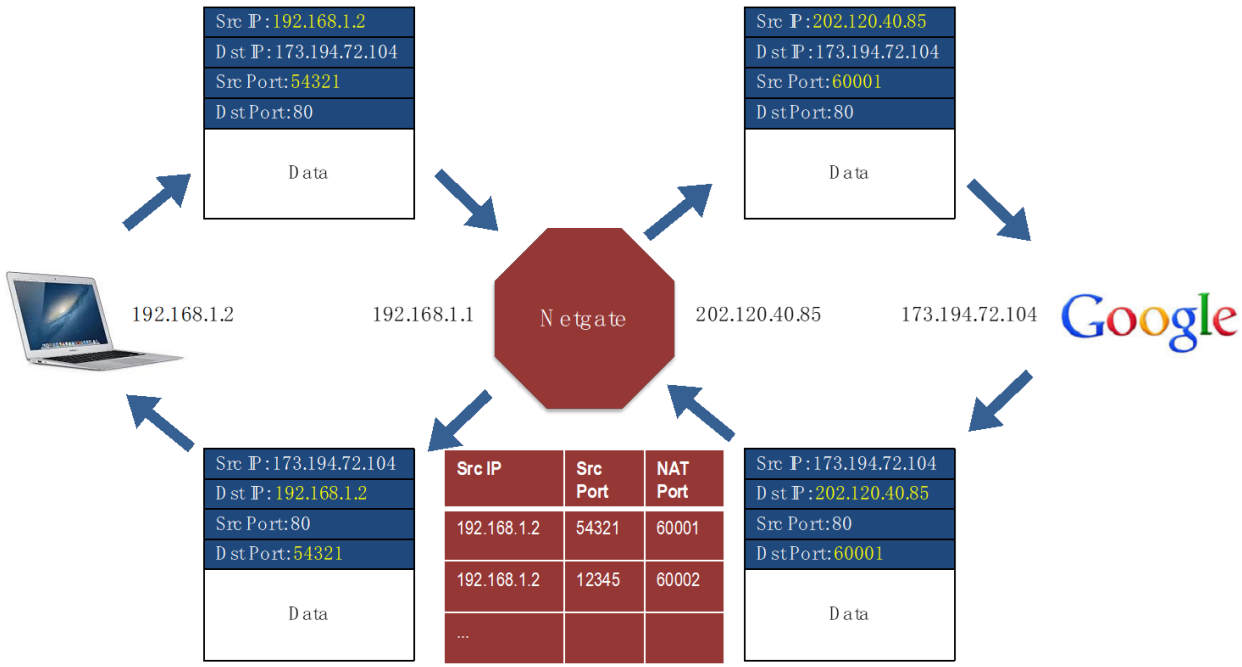
这样的话整个路由表就小了很多，很方便我们去做scalability。

我们拿到ip先和掩码做一个and，如果结果和18.0.0.0匹配，那么就从1号网口发出去。所以如果我们实验室内网访问，那么就直接根据1和2网口，内部转发即可。如果是一个乱七八糟的IP，那么我们就要通过3号的外网路由出去。这样就可以减少路由表的大小。

4. NAT（Network Address Translation）：内网和外网

NAT（network address translation 网络地址翻译）是因为电脑太多了，网络太少了，所以要分为内网和外网。内网开头是10.和192.。

以10地址为例，出了内网，这个10地址是没有意义的，当我们要连到外面地址的时候，要把ip做一个转换。比如我们请求百度，百度的destination是什么呢？肯定是学校出口公网ip，那么学校里那么多机器，我们就要根据端口号做一个转发。



路由器有对内ip 192.168.1.1和对外ip 202.120.40.85，负责对内对外的转换。负责NAT port和内网IP:端口的映射。这里是有好几个问题的。

1. NAT Port的数量是有限的；
2. 这是典型的破坏了层级的设计。Ip地址不够了，我们用到了上层end-to-end layer的port的概念。我们用上层的概念解决了下层的问题，所以这并不是一个很漂亮的设计。而且必须要求上层必须是port，如果我们用的不是TCP/UDP的port概念，那么NAT就不能用了。

Take Away Messages

1. 网络层解决的问题是去中心化的设计里，并没有一个全局的组织来统计哪里到哪里是通的，所以我们需要在局部构成一个网络体系，和其他连在一起构成一个整体。
    b. 在做advertisement的时候，并没有任何机制保证advertisement是真的，如果我们对外advertisement的信息是错误的，对方是没有判断机制的。
    c. 在形成路由的过程中，分为link-state和distance-vector两种方法。这两种模式，后者肯定是比前者更适合大规模的网络，但是依然不够。所以我们提出了path-vector，我们要把整条路径都放在advertisement里，这样的好处就是不会出现loop的情况。

d. 但是这样还是不够的。因为量太大了，所以我们引入了hierarchy，把一堆节点拼在一起变成一个region。在互联网上，这叫做自治系统（AS），它对外和对内分别负责包的forward，这样就可以大大降低路由表的数量，通过掩码就可以把255项变为一项，这意味着查表的速度可以更快。但是在一个非常高速的路由器里面，要做到这么快还是挺难的，因为每次查表我们只容许一两千个cycle。

e. 这时候我们就需要在Data Plane上提升查表的速度，如果我们能把表做一个很好的组织，使得最常用的项在L1 Cache中，这样我们查表的时候一个cycle就可以搞定，那么偶尔查L2，L3和内存，对整体的查表速度也没有太大影响。

2. NAT机制：这和我们在讲自治区域的时候也是类似的。因为电脑太多了，网络太少了，所以要分为内网和外网。比如学校网对外只有3~5个节点，那么所有访问外网的请求都会先发到3~5个边界节点。它解决的事情是网络IP不够了，这样我们就可以让内部变成10.打头或者192.168.打头。为了记录下对外的映射，我们要在路由器里记录下对外的端口，到时候发回数据的时候再映射回内网的IP和端口。

4. Network-end2end layer: best effort is not enough

接下来我们就要讲更上层的end-to-end layer。网络层到现在为止，依然没有一些绝对的保证，比如delay。这不像我们之前说到的打电话的网络，我们把连接预留一小段。而且网络层还不能保证网络包的order是一致的。甚至连包能不能到都不能保证，也有可能包的错的，或者包发错了人。所以整个网络非常地不靠谱，这才是我们需要end-to-end layer存在的理由，因为没有一个单独的设计可以解决所有的问题。

1. UDP：纯粹发包，不管丢包，和IP协议相比只是多了端口号，有了端口号一个IP就可以复用给不同的应用程序。
2. TCP：保证数据是有order的、没有丢包、也不会有包的重复，也可以做流控flow control。

我们总共要实现七个需求：

Assurance of End-to-end Protocol

1. **Assurance of at-least-once delivery**
2. **Assurance of at-most-once delivery**
3. **Assurance of data integrity**
4. **Assurance of stream order & closing of connections**
5. **Assurance of jitter control**
6. **Assurance of authenticity and privacy**
7. **Assurance of end-to-end performance**

1. At least once

确保at least once，我们要设置一个RTT时间（发过去的时间+处理时间+返回时间ack），超过RTT这个时间我们再重发这个包。

需要做如下事情：

1. 发包的时候放一个nonce（一个网络上的包的唯一的标识），不能发完就把包删了
2. 如果发现timeout，就要resend。
3. Receiver要把nonce返回给sender
4. 我们不能无限制地去retry，我们最多重试3~5次，如果还是不行就往上报错。**【有限次数的重试】**

At least once，要面对的就是两难问题。

1. 数据没送到
2. 数据收到了但是没有收到ACK。

对于at least once来说，是没有一个absolute的guarantee的。我们只能保证a和b之间如果有一条通道是通的，我们最终可以保证收到这个请求。

如何确定超时？这很关键。

- 第一种方法是fixed timer，设置超时时间1s等，这种太短就是无谓重试，太长就是浪费时间。并且fixed timer会导致collapse，举个例子，当我们在收邮件的时候，大家都有邮件的客户端。邮件客户端有一个特点，它要设置比如5分钟收一次，收到一次以后改成1分钟收一次。对于这种邮箱来说，可能到固定时间以后整个局域网都在收邮件。比如云服务器厂商每到晚上两点，服务器就crash了，服务器的硬盘在狂转，因为在杀毒。

> **Fixed timer is evil.**
>
> 网件路由器代码是用来同步时间的，写死了美国威斯康辛大学的SNTP的对时服务（地址写死了，fixed timer，如果获取时间失败，就会一直1秒钟一次来at least once来获取时间）。如果有问题就会重试，每秒尝试一次。然后整个大学就一直在承受路由器的ddos攻击。所以fixed timer我们要非常小心。
>
> **SNTP中其实有一个部分叫go away，我这里塞住了，你给我滚，别来给我发消息了。威斯康辛没有实现这个feature。**

- 那么既然fixed timer有问题，我们能不能设置一个adaptive timer呢？比如我们把超时时间设置为RTT的150%，并且会指数型的设置超时时间。比如1s->2s->4s->8s。
- 没有超时而误判为超时的情况是非常常见的。

- 我们有没有不依赖时间的方法呢？我们可以用NAK方法。它的逻辑是反过来的，我们发一个3号包，对方和你说3号包没收到重发一下（而不是告诉你哪些包收到了），这样我们就能100%确定对方确实没收到。这样我们就可以直接把那些丢掉的包发回去。在实际时间的时候，NAK也要考虑很多问题。

2. At most once

在at least once中，sender要记录某些包收到了还是没收到的，如果误判了超时，那就可能发了duplication。所以at most once需要记录哪些包收到过，一旦我们记录了一个table，它就可能变得非常大，导致空间和搜索时间都变长了。我们需要用到幂等和其他方式去解决。

总之，at most once非常难，有很多trade off。

3. Data integrity

完整就是数据没有被改来改去。Receiver收到的内容必须是和sender发的是一样的。一个基本做法就是checksum。我们在之前说道，在link layer已经有checksum了，为什么我们在end-to-end layer还要做checksum。因为link layer的海明编码一旦错了2~3位结果可能是错的。所以底层的不一定那么可靠，并且网卡检测完checksum复制到内存的时候，也有可能出错。所以end-to-end layer再加一层检测就可以增加安全性。又可能我们的包误发到别人那里去了，所以也不能依赖于底层的checksum。

4. Stream order & closing of connections

比如我们传包的时候说，我正在传7个分片中的第3个。当收到了之后，顺序可能会混在一起。所以我们需要预留一个足够大的可以容纳整个message的buffer。这样整个message就可以还原出来。一旦发生了out-of-order，分为几种情况：

1. 不允许发生乱序的情况，收到1,2号包之后，丢弃收到的4号包，必须等3号包。这样浪费带宽。
2. 收到包之后，先放到buffer里，等到in order的时候，我再统一把全部的包收上去。但是如果有一个package是坏的。这个buffer得一直维护住。这个对于receiver来说也是一个很大的内存压力。

所以最终我们combine一下两种方法，如果buffer满了但是还有包没有收到，那么我们只能丢掉buffer重传。

所以out-of-order的关键问题在于我们要用一个多大的buffer去对数据排序，如果我们receiver无原则的容忍sender。Sender可以利用这一点来消耗receiver的资源。

5. Jitter control 抖动控制

当我们看视频的时候，我们发现视频动不动就卡一下，动不动卡一下，我们用什么方法？

就是暂停一下，也就是相当于buffer一下。

Buffer就是用来对抗jitter的。缓冲的时间怎么确定呢？我们可以用这个简单的公式来确定我们要缓冲多少个segment。换句话说，我们最长的一次delay是100ms，最短是9ms，平均是10ms，那么我们应该缓存 9.1个包((100-9)/10)。我们收到一个包开始看视频，哪怕遇到了100ms也没关系，因为我们

已经缓冲了9个包，可以抵御一次D_long。所以对抗jitter的方法就是缓存。但是游戏就没有办法用这种方法了。Jitter control实际上是不同类型的包有不同的要求。

6. Authenticity and privacy

我们需要对网络加密，核心就是加密的公钥私钥体系。

公钥和私钥其实是人为分开的，在数学上它们的地位是一样的（可以用一个密钥加密，另一个密钥解密，也可以反过来；只是我们使用的时候选择公开了两个密钥中的某一个，它就被称为了“公钥”），只是我们在使用的时候公开了一个而已。

我们来举一个例子：

1. 假如我现在要给张三发一个消息，我希望这条消息只有张三能够看见；我就用张三的公钥给这个消息加密；加密之后，我可以放到微博上、朋友圈里、区块链上（任何公开的地方），只有有私钥的张三才能够看到里面的内容。【隐私信息的传递】
2. 反过来，如果有一个很重要的人物，他说一句话，所有人都要相信这句话是他说的；这个时候他要做的事情就是用自己的私钥，把他说的话来做一个签名——所谓的签名，用最简单的逻辑来理解，就是用私钥来加密。我如果看到这句话，我要检验它是谁说的，我就用这个人的公钥来做个解密，发现就是那句话，那就是那个人说的。【让所有人都知道这句话是我说的】

7. end2end performance

end-to-end performance是希望发包的速度足够快，但是发包太快可能会拥塞整个网络。发包的速度不是决定网络瓶颈的点，而是网络本身带宽的限制，以及接收者能不能接收这个频率的发包速度。

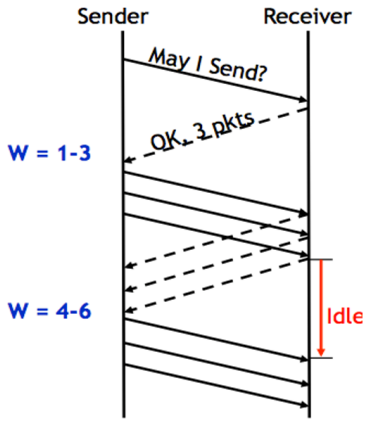
怎么让sender发包充满整个网络

发送者发一个segment（包的碎片），然后receiver返回ack，然后再发下一个。这可以很好地满足准确性，一旦收不到就重新发。但是这毫无疑问很慢，大量时间都在等待ACK。所以这个方法很慢。

一种直观的想法是pipeline，不管你收没收到，我先一直发。Pipeline打破了sender和receiver之间的关系，没有做等待，这就会导致太快，可能导致网络包和ack被丢弃。

我们能不能让sender保持一个已经发送的segment 的list，如果list中全部ack了，那再往下发一个list。这就是在lock step和pipeline之间的权衡。n怎么确定呢？比如我们要发n个包，实际上这个数字是由receiver的buffer来决定的。

Fixed Window



Receiver tells the sender a window size

Sender sends window

Receiver acks each packet as before

Window advances when all packets in previous window are acked

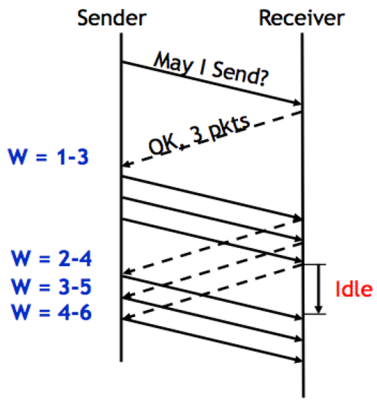
- E.g., packets 4-6 sent, after 1-3 ack'd

If a packet times out -> resend packets

Still much idle time

这个n个包的块就叫做window，receiver是在这个ack和下个ack之间差的时间还比较长。这个idle期间的buffer是没用的。我们就觉得这段时间可能是可以优化的。在这段时间之内，sender有没有可能再发一些包。当sender收到第一个ack的时候，是不是能发第101个包？（每收到一个包，窗口往后滑一个）

Sliding Window



Sender advances the window by 1 for each in-sequence ACK it receives

- Reduces idle periods
- Pipelining idea

But what's the correct value for the window?

- We'll revisit this question
- First, we need to understand windows

我们这个window size应该多大呢？

Sliding Window Size

window size ≥ round-trip time × bottleneck data rate

Sliding window with one segment in size

– Data rate is window size / RTT

Enlarge window size to bottleneck data rate

– Data rate is window size / RTT

Enlarge window size further

– Data rate is still bottleneck

– Larger window makes no sense

- Receive 500 KBps
- Sender 1 MBps
- RTT 70ms
- A segment carries 0.5 KB

- Sliding window size = 35KB (70 segment)

Load Shedding: Setting Window Size

For performance:

– window size ≥ round-trip time × bottleneck data rate

For congestion control:

– window size ≤ min(RTT x bottleneck data rate, Receiver buffer)

– Congestion window

2 windows become 1

– to achieve best performance and avoid congestion

我们假设receiver可以收500k，sender以1兆的速度来发包；RTT就是从sender到receiver，一个来回要70ms，每一个segment是0.5k（每一个包是0.5k）。这里bottleneck data rate就是500kb（有效的带宽，当然是两根水管里面更窄一点的那个）。

我们把bottleneck data rate可以看做一根管子的截面积，而round-trip time是管子的长度。而window size就是不等收到ack前还要继续发的数量，我们希望管子中都是我们扔出去的纸飞机。我们希望receiver收到一个包，sender就发送一个包，并且此时管子中（网络中）充满了包。

那么我们在整个滑动窗口的过程中，sender不知道bottleneck多少，但是它正好卡在bottleneck的角度去发送，一旦slide window都填满了，就不能再发送了，receiver产生ack的速度也不会比网络发送的速度更快。但是我们依旧要估算这个RTT，在Linux kernel中通常是通过3/4加权来迭代计算RTT。

控制congestion

大量的包同时出现在网络上导致大量的包出现丢失。一旦网络上的包太多，导致排队的包还没有处理的时候就导致超时重发了。此时的排队就没有意义了。所以交换机单纯地变大，只要我们有超时机制，只会导致更多的排队超时，并不能解决问题。

Congestion window应该是慢慢地变大的，如果没有发生congestion，就继续上涨，如果发生了，就立即下降。

在TCP中用到了一个AIMD算法，这是一个比较保守的算法。但是AIMD有一个问题，就是开始的时候非常慢。我们可以在开始的时候用指数级的方式去增加，这就是Slow Start阶段（理论上应该叫fast start）。

Retrofitting TCP

1. Slow start: one packet at first, then double until

- Sender reaches the window size suggested by the receiver
- All the available data has been dispatched
- Sender detects that a packet it sent has been discarded

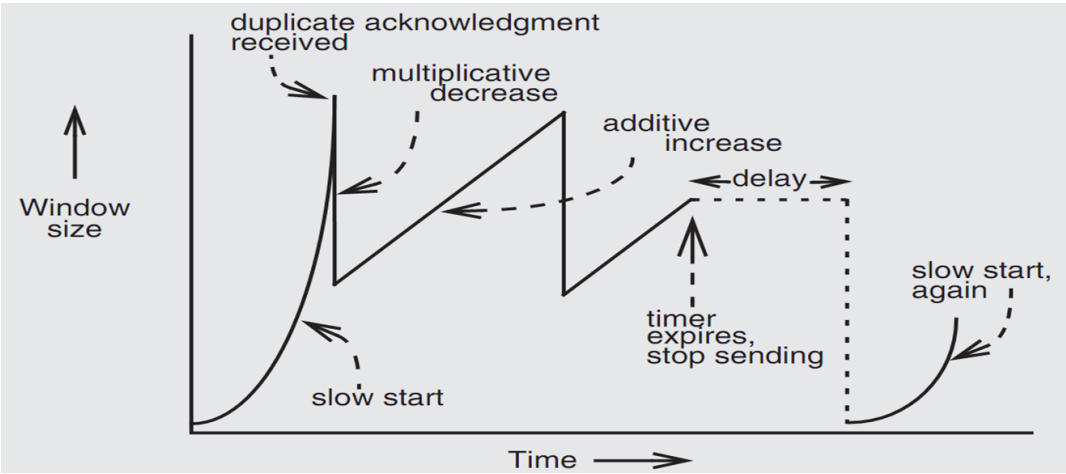
2. Duplicate ACK

- When receiver gets an out-of-order packet, it sends back a duplicate of latest ACK

3. Equilibrium

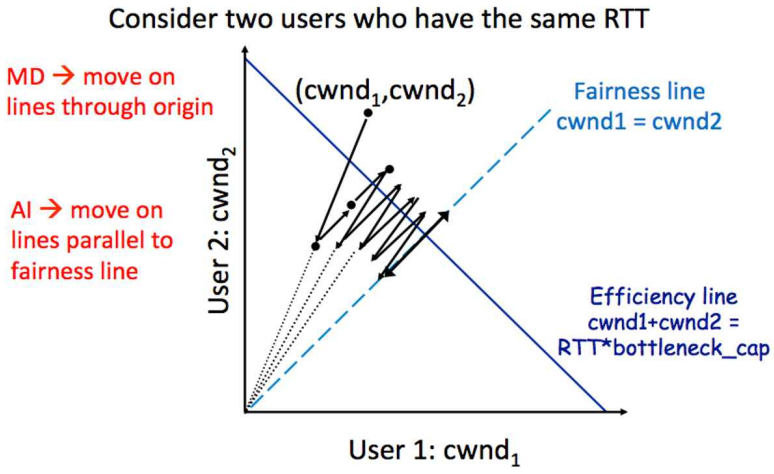
- Additive increase & multiplicative decrease

4. Restart, after waiting a short time



这个就包括了slow start、丢包腰斩阶段，重新slow start阶段。很可能极限就在红线为止。但是红线下的锯齿状空白部分（没有被积分到的位置）都是被浪费掉的空间。这个可以很好地实现公平性。

AIMD Leads to Efficiency and Fairness

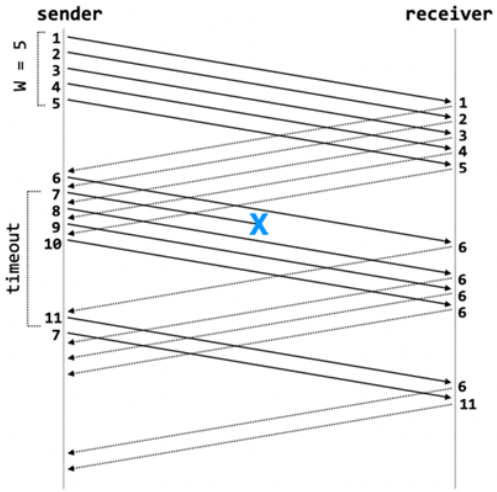


80

TCP Fast Retransmit / Fast Recovery

When a sender receives an ACK with sequence number X, and then three duplicates of that packet, it immediately retransmits packet X+1

- Example: sender receives 5 6 6 6 6
- Infers that packet 6 is lost, immediately retransmits
- On fast-retransmit, window decrease is as before: W = W/2



8. DNS

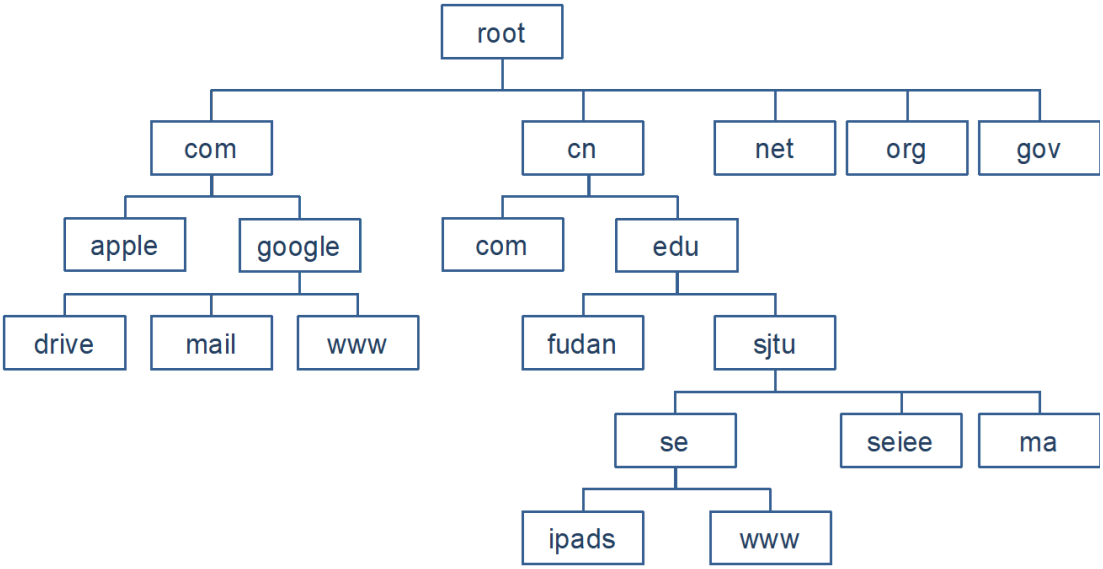
DNS（domain name service），它也是在end-to-end layer中的，在网络中可以把IP藏起来。当我们去访问一个域名www.sjtu.edu.cn的时候，最终可以解析为IP(202.120.2.119)，这就是一个look up的过程。

IP在网络上其实已经够用了，但是我们希望有human-friendly的方式。一个域名可以有多个IP，这样用户端就可以挑一个距离近的。一个IP也可以对应多个域名，挂载不同的端口上即可。

映射关系查找算法

1. 每个人维护一个hosts.txt，把域名和IP写下来，就和我们的目录一模一样。但是毫无疑问这个算法是不够scale的，文件会越来越大。所以在1984年，出现了BIND(Berkeley Internet Name Domain)，这个系统今天还在用。

2. Binding一定不能做成一个中心化的服务器，因为全世界这么多人在访问，但是我们依然要把数据存在很多个服务器上，只做提供name service的服务。我们怎么样才能组织这么多的域名服务器。我们首先分成一个个zone，然后产生很多个name server，最root的这些域名是由ICANN这么一个组织来维护。通过这种方式，做了一个分工的解耦。如果我们要注册一个cse.sjtu.edu.cn要层层审批到ICANN，那就太麻烦了。去中心化就是放权到子zone中。



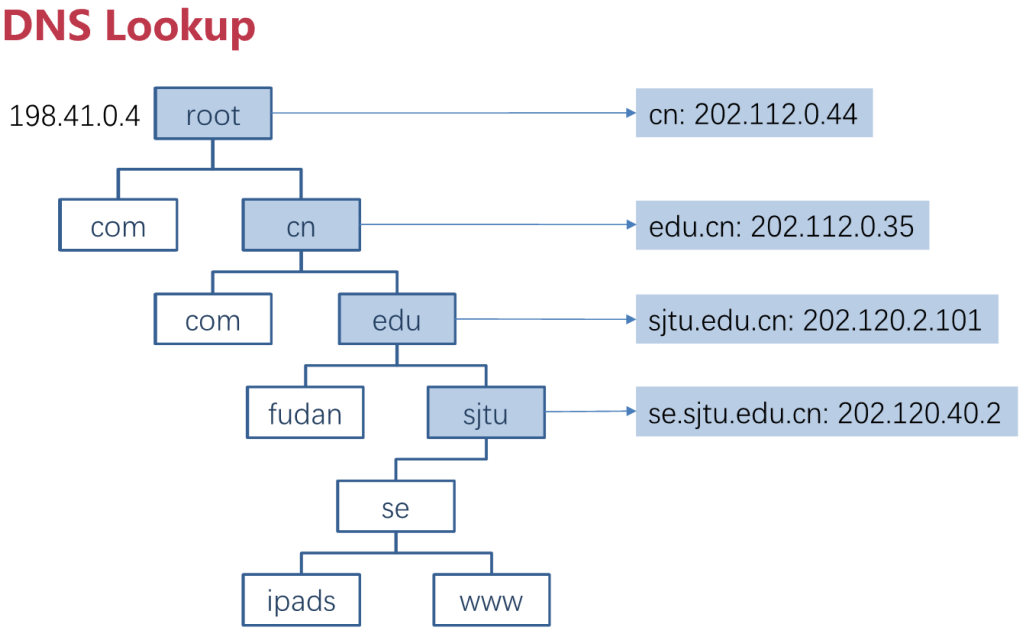
Basic DNS Look-up Algorithm

Example: lookup IP of "ipads.se.sjtu.edu.cn"

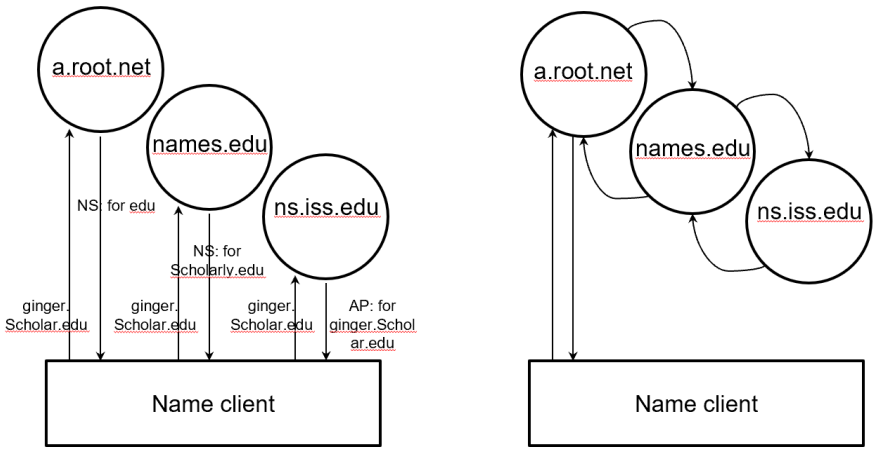
Traverse the name hierarchy from the root

- The root will tell us the "cn" name server IP,
- which will tell us the "edu.cn" name server IP,
- which will tell us the "sjtu.edu.cn" name server IP,
- which will tell us the "se.sjtu.edu.cn" name server IP,
- which finally tells us the "ipads.se.sjtu.edu.cn" IP

Such algorithm is called delegation



怎么做fault-tolerant呢?每一个zone都有多台服务器，一个挂了就问另一个。如果每一次都会问root的话，那root server会先crash掉。所以在现在的场景下，我们在游览器里按下回车的时候，初始的DNS可以到任何一个name server，比如先问交大的DNS。如果我们不知道交大的DNS是多少，这是不可能的，因为在分配IP的时候，就会知道交大的DNS。Google给全世界提供8.8.8.8，还有114.114.114.114，这样我们可以加进/etc/hosts。如果没有的话，那就是我们第一次连到无线网的时候，会拿到一个交大的DNS服务器IP。直到没有人知道才会找root。



第二种的情况就是可以cache住，对于交大的DNS服务器来说比较合适。但是这种有一个问题，因为第一种是无状态的，第二种是有状态的，如果服务的人太多了，消耗的资源就会更多，所以在具有同样资源的情况下，recursion的方法能够服务的人会比较少。

无状态的情况下，所有的状态保存在name client这边；从系统设计的角度来看，我们希望的是无状态——意味着发生任何的错误，重启之后，对于client是无感的。

我们讲完DNS的设计之后，我们要来看一下在DNS设计的背后做对了哪些事情。

1. DNS利用hierarchy，把责任分割成了zone（各级人员承担不同层级的责任，避免了单点故障），这样做到了去中心化（限制了不同模块之间的交互，比如.com和.cn，互联网最早的时候就是去中心化的思路，不希望有一个强大的组织控制互联网）。
2. Global name很好地避免了对context的要求（再比如邮箱：zhangsan@sjtu.edu.cn，邮箱就是很典型的自带context，把context直接embed到里面去，这个XX@XX全世界就是唯一的）。并且cache可以减少query的量，还可以通过delegation来分配到多个。管理起来也很方便。
3. Fault-tolerant也很重要，因为每个DNS Server有多个。

缺点就是管理的人到底是谁，政府还是NGO，能不能给犯罪团伙提供服务，这都是有很多争论的。

第二个就是对于root server来说还是有很大的漏洞，我们讲数据库的时候，一旦cache hit就直接拿到了，但是如果我们查找一个不存在的东西，一定会cache miss。如果我们找一个不存在的域名，那么所有DNS服务器都不知道，就要问根服务器，这就可能成为一个DoS攻击（所以我们在进入根服务器之前，就要做一些筛选）。

Take away message

1. 从网络整个层次角度来看，底层专业知识更多，上层软件知识更多。上层的协议是用来解决各种各样的问题。学习协议的最好方法就是读手册看什么情况下会发送什么样的数据包。但是为什么要这么设计，背后有很多理论性的知识。
    a. 底层的IP层，包括底下的link layer不能解决at most/least once、数据的完整性、数据的顺序、jitter control。
    b. 因此，我们需要上层end2end来为我们做更多的guarantee。