

## 目录

2022/2/15.....	3
2022/2/18.....	6
2022/2/22.....	11
通过计数方式来分包.....	13
错误识别和改正.....	15
CRC（循环冗余校验）.....	17
Mac 层协议.....	19
2022/2/25 (NO) .....	27
IP Packets, IPv6 & NAT.....	34
2022/3/1 (NO) .....	36
2022/3/4.....	36
DC TOPOLOGY.....	38
三层的交换机制 fan-out (可扩展性) .....	39
CLOS Network.....	39
便宜路由器的胖树的目标.....	40
Portland Address.....	41
2022/3/8.....	45
TCP.....	47
2022/3/11.....	48
QoS.....	55
RED.....	61
2022/3/15(最后 15 分钟).....	63
RED(Random Early Drop).....	65
QoS principle.....	69
2022/3/18.....	79
了解 DPDK.....	79
2022/3/22.....	94
Cryptography (密码学) and HTTPS.....	94
HTTPS.....	105
2022/3/25(No).....	107
2022/3/29.....	115
可选功能.....	119
容器虚拟化.....	122
容器镜像的设计和实现.....	126
Dockerfile.....	128
容器网络.....	130
VOLUME 与容器持久化.....	133
容器安全.....	135
2022/4/1.....	138
KataContainer.....	140
OCI Spec.....	141
Runc.....	142
轻量级 VM 虚拟化.....	145

Google gVisor.....	148
KVM.....	153
FireCracker.....	155
2022/4/8.....	155
FireCracker.....	157
FireCracker VMM.....	160
K8s.....	162
K8s 的架构和组件.....	164
K8s Pod.....	166
Pod 的生命周期.....	168
Replication Controller.....	169
API 对象.....	170
2022/4/12.....	173
声明式和命令式的管理方式.....	175
命令式和声明式的比较.....	177
Kubectl (K8s 的命令行工具) .....	178
从 Docker 命令来看 kubectl.....	179
kubectl 管理对象.....	179
kubectl 和 K8s 集群内部的交互.....	182
API 组 (Groups) .....	184
对象模型 (Object Model) .....	185
对象表达: YAML.....	186
Workload (负载) .....	187
K8s 的核心对象.....	188
命名空间 (Namespaces) .....	189
Pod.....	190
Labels.....	190
选择器: Selectors.....	191
Service.....	192
2022/4/15.....	193
Services.....	195
ClusterIP Service.....	198
NodePort Service.....	199
LoadBalancer Service.....	200
ExternalName Service.....	201
workload 对象.....	202
ReplicaSet.....	203
Deployment.....	204
DaemonSet.....	207
StatefulSet.....	208
2022/4/19.....	211
Job.....	213
CronJob.....	214
K8s 的网络.....	216

Pod 内容器间的通信.....	217
Pod 间的通信.....	219
CNI 模型（Container Network Interface） .....	221
K8s 中的存储.....	223
Volumes.....	224
持久化卷（PV） .....	225
PVC.....	226
StorgaeClass（存储类） .....	228
怎么实现一个 Service？ .....	230
2022/4/22.....	232
K8s 的组件及其交互方式.....	235
集群状态和 etcd.....	237
API Server.....	241
Scheduler.....	246
Kubelet.....	249
2022/4/24.....	251
Kubelet.....	254
启动一个 Pod 的工作流.....	256
Controller.....	257
Node Controller.....	259
创建一个 Deployment 的工作流.....	260
Informer.....	261
Workqueue.....	263
Lab 相关.....	268
2022/4/26.....	268
扩展 K8s：CRD 和 K8s 插件.....	268
K8s 总结.....	270
Google BORG：K8s 的前身.....	273

## 2022/2/15

这个开销意味着什么？网络把 CPU 全部烧光了。买一个几十核的虚拟机一个月要上万。以前小公司建立机房得有空间、电费、运维人员、空调等，现在这些费用都没有了。

我们要做些什么事情：了解路由的协议、有 QS、有纠错重传的功能，不同的数据包是有优先级的。200G 以后的网络，约等于内存的性能，所以 CPU 和 memory 现在可以跨物理节点来访问，并且没有停止增长的趋势，因为光纤可以打无数条光，没有上限。现在是网络可以烧掉服务器大量的计算资源，而且完全是底层的数据解析的功能。

怎么把网络带宽提上来，如果 CPU 不够，那么网络带宽就被浪费了，就不能达到应有的性能了。我们要知道网络是怎么传的、怎么交互的，对于一些后期 message queue 等，和 network system 怎么进行交互。我们要用最小的计算资源来处理好网络并且把计算资源尽快发出去。网络的性能最终就体现了总体业务的能力。

11 次课，3 个 LAB，平时分 34 分，上课可能随机抽人，约 3~5 分，一次不来扣 1 分。

- Computer Networking A Top-Down Approach Featuring the Internet
  - NETWORKS Design and Management, Steven T. Karris
  - Computer Networks: A Systems Approach, Larry Peterson and Bruce Davie.
- Computer Networks (5th Edition) Andrew S. Tanenbaum, David J. Wetherall

## Lab Contents and Policy

### ■ Lab goal

- Experience on transport scheme.
- Experience on TCP traffic control
- Networking HW/SW Acceleration (NFV)

### ■ Working together is important but without Reusing

- Discuss course material in general terms
- Work together on program debugging, ..
- Final submission must be your own work

1. TCP alike System: Go-Back-N, Selective Repeat, TCP rdt protocol
2. DPDK Snd/Rev BASIC
3. QoS Router design and implementation with DPDK

首先我们要做 TCP 的交互重传，对于任意丢、任意乱序的包怎么做重传。TCP 更重要地是做 QoS（服务质量），和快递一样，有些包是 24 小时到，有些包是 48 小时到，取决于包的优先级。在中间我们就要研究面向业界的网络协议栈。传统的 socket 在业界已经没有人在用了。socket 是在 OS 内核的。而 DPDK 是用户态的。为什么要分内核态和用户态的网络协议栈，效率是有本质的变化的。OS 的网络调用是方便，但是缺点是性能不高，而用户态的网络协议就是高速的。

## Class Contents

### ④ Layering of Network

- MAC layer, collision control
- IP layer
- TCP layer
- QoS and congestion control
- HTTPS

### ④ Applications, novel protocols and emerging technologies: DPDK, QAT and NFV

### ④ Acknowledges: CMU 15-441, Intel NPG.

我们从物理层不断往上爬。网上最终是安全性。课程对标 CMU 15-441, Intel NPG。我们的实验也会用到开源的代码。

这周是用 C++ 调通 UDP，做一个 client-server。看一下 UDP 的 packet 的概念和 API。

## What is the Objective of Networking?

- Enable communication between applications on different computers
  - Web
  - Peer to Peer
  - Audio/Video
  - Funky research stuff: **distributed sys., data center, cloud computing, sensor network, IoT, ubiquitous comp., etc.**
  - **DPDK, ASIC Cloud, SDN/NFV**
- Must understand application needs/demands
  - Traffic data rate
  - Traffic pattern (bursty or constant bit rate)
  - Traffic target (multipoint or single destination, mobile or fixed)
  - Delay sensitivity
  - Loss sensitivity

## What Is a Network?

- Collection of nodes and links that connect them
- This is vague. Why? Consider different networks:
  - Internet
  - Andrew
  - Telephone
  - Your house
  - Others – sensor nets, cell phones, ...
- **Class focuses on Internet, but explores important common issues and challenges**

Andrew 是 CMU 做的一个局域网。电话宏观上也是一个网络。

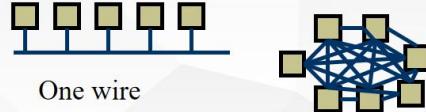
## Basic Building Block: Links



- Electrical questions
  - Voltage, frequency, ...
  - Wired or wireless?
- Link-layer issues: How to send data?
  - When to talk – can either side talk at once?
  - What to say – low-level format?
- Okay... what about more nodes?  
What about with high speed NIC, 100G-200G?  
[https://github.com/codecat007/dpdk\\_doc](https://github.com/codecat007/dpdk_doc) (for beginner)

Node 可能是一个路由器，可能是一个电脑。我们会从物理层讲一讲这个东西。我们更主要讲 link-layer：网络怎么连上的。为什么用户态？为什么高效？可以打开 [github.com/codecat007/dpdk\\_doc](https://github.com/codecat007/dpdk_doc) 看看。

- ... But what if we want more hosts?



Wires for everybody!

- Scalability?!

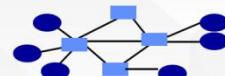
\*网络拓扑：汇聚层，业务的编排管理也是有说法的。比如两个业务紧耦合的情况下，一定要尽可能在网络上编排的近一些。

### Local Area Networks (LANs)

- Benefits of being “local”:
  - Lower cost
  - Short distance = faster links, low latency
    - Efficiency less pressing
  - One management domain
  - More homogenous
- Examples:
  - Ethernet
  - Token ring, FDDI
  - 802.11 wireless

### Multiplexing

- Need to share network resources



- How? Switched network
  - Party “A” gets resources sometimes
  - Party “B” gets them sometimes
  - Interior nodes act as “Switches”
- What mechanisms to share resources?

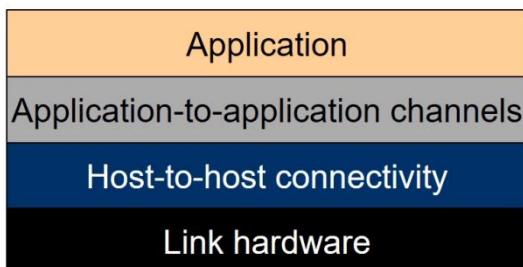
怎么去传呢？在不同节点之间有很多种路径，这就是路由算法的核心。

15:00

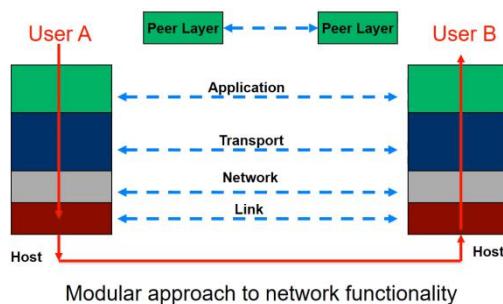
邮递员不管前面是经过几个邮递员中转的，只负责最后投递到信箱里。所以我们每次只关心一步。比如说在路由器，数据包在一个 buffer 中，里面有大有小，里面包含了不同设备的数据包，所以就要通过 multi-plexing 来分发。包括说路由器的 buffer 如果满了就会丢包。电话的优先级可能就高于看电影、看电影的优先级可能又高于文件传输。所以我们是否要对数据包做分类地处理，这都是我们要考虑的事情。

**2022/2/18**

今天我们就开始要把大的架构搭出来。除了以太网，其他的协议我们只会介绍一下。我们需要知道协议层的多样性、一致性。我们会介绍一些国际标准的模型。我们需要它们这些模型设计的理论和标准。



上面这张图就是 layer 的例子，每一层做自己的事情。有点像流水线，每层做自己的事情。



这是大的网络架构。还有的就是 peer-to-peer，也就是在网络的基础架构上再搭建起几层虚拟的网络。它为什么要 overlay，就是它是在物理节点上平铺的。

爬格子就有上下的区别，所以每一个网络服务会依赖于底层并且向上提供一个服务。这样的好处就是黑盒的形式，上层和下层都不需要关注我们是怎么实现的。另一个好处就是可组装，网络可以切换、做整体的更新都和我们的应用无关。

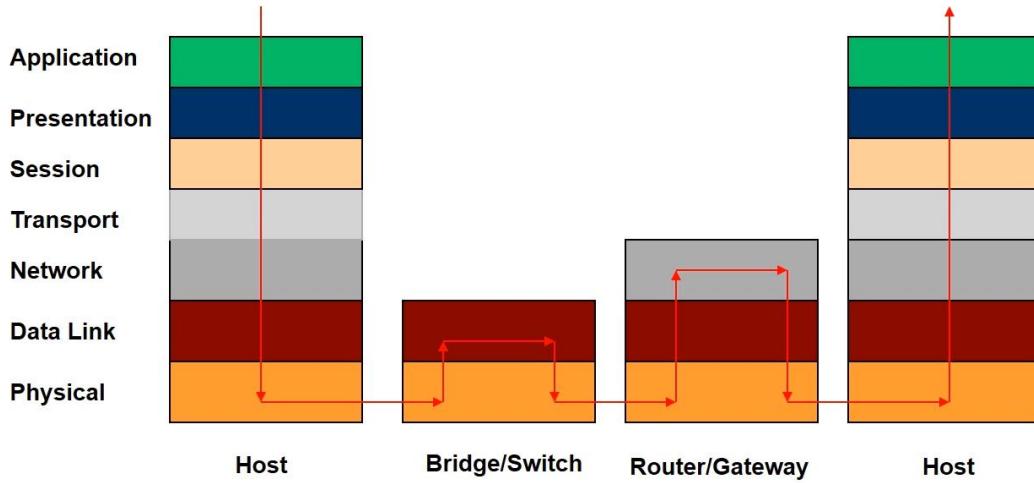
层和层之间的好处就是标准化之后，就可以自由组合。但是层和层之间也没有那么通用，比如 windows、Linux、unix 之间的网络协议栈是不通用的。由于它们不通用，所以 APP 是要在不同平台上的协议栈上重新开发的。

我们需要知道网络的部分是怎么交互的，网络的语法语义比较抽象。

## OSI Model: 7 Protocol Layers

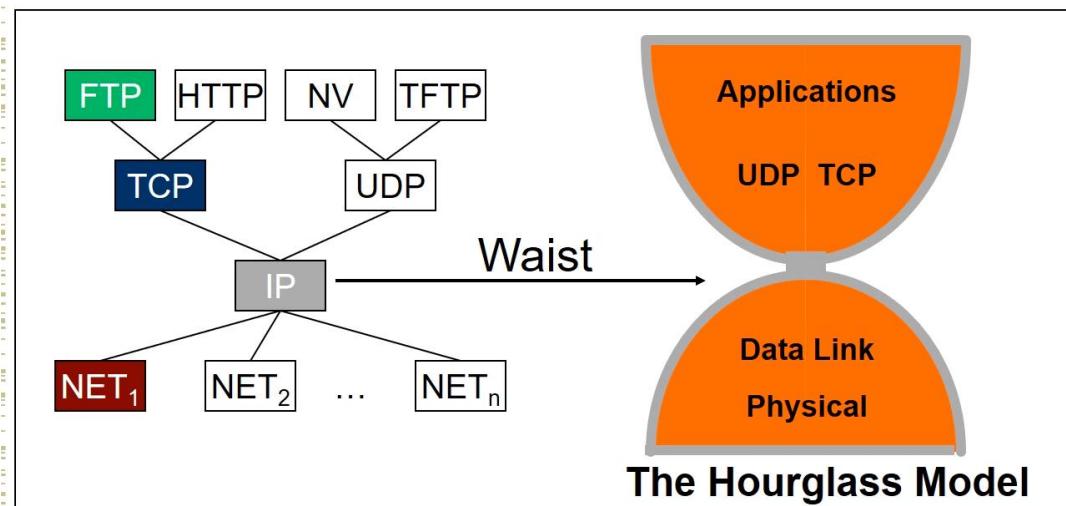
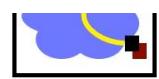
- **Physical:** how to transmit bits
- **Data link:** how to transmit frames
- **Network:** how to route packets
- **Transport:** how to send packets end2end
- **Session:** how to tie flows together
- **Presentation:** byte ordering, security
- **Application:** everything else

# OSI Layers and Locations



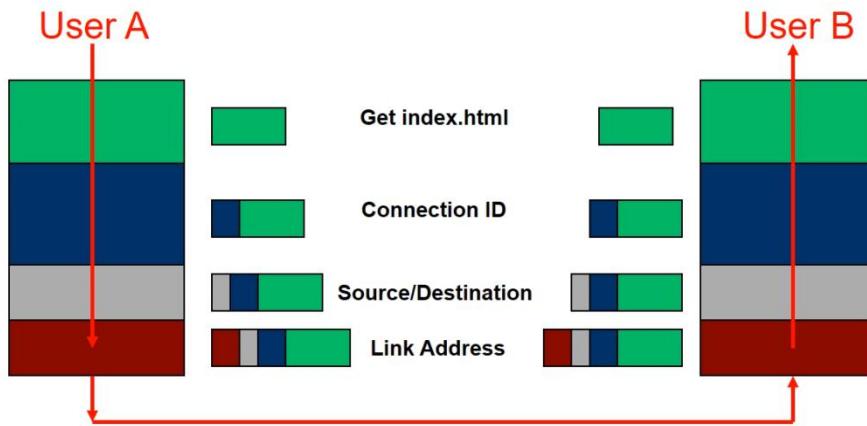
上图是在端到端的情况下的交互过程。Bridge 和 Router 的区别就是：二层交换、三层路由。前者到连接层就结束了，后者还要做 network 的事情。传输过程就是这样，传输过程中经过了几个交换机和路由器我们是不知道的，我们只知道第一跳发给了谁。

## The Internet Protocol Suite



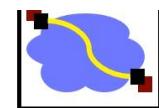
### The waist facilitates interoperability

第三个就是协议栈，每一层在做什么事情。往上的应用层千变万化，主要两大传输层的协议就是 TCP 和 UDP。IP 是“瘦腰”，把上下的复杂性都屏蔽掉了。所以这是一个沙漏模型。

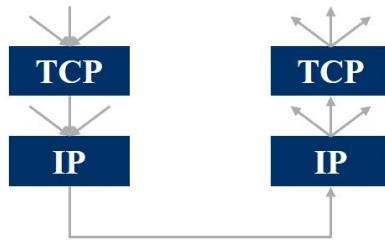


网络整体在做的事情就是“发快递”，商店到快递站一层层打包（如木架子、塑料布、电动车盒子），最终我们一层层拆开盒子就得到了我们买的手机。

## Multiplexing and Demultiplexing



- There may be multiple implementations of each layer.
  - How does the receiver know what version of a layer to use?
- Each header includes a demultiplexing field that is used to identify the next layer.
  - Filled in by the sender
  - Used by the receiver
- Multiplexing occurs at multiple layers. E.g., IP, TCP, ...

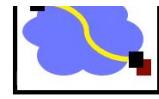


V/HL	TOS	Length
ID		Flags/Offset
TTL	Prot.	H. Checksum
Source IP address		
Destination IP address		
Options..		

IP 地址当时就是 32 位地址，每一位有自己的意义。

我们一个网络要实现什么事情呢？

# Goals [Clark88]



## 0 Connect existing networks

initially ARPANET (Advanced Research Projects Agency Network) and ARPA packet radio network

### 1. Survivability

ensure communication service even in the presence of network and router failures

### 2. Support multiple types of services

### 3. Must accommodate a variety of networks

### 4. Allow distributed management

### 5. Allow host attachment with a low level of effort

### 6. Be cost effective

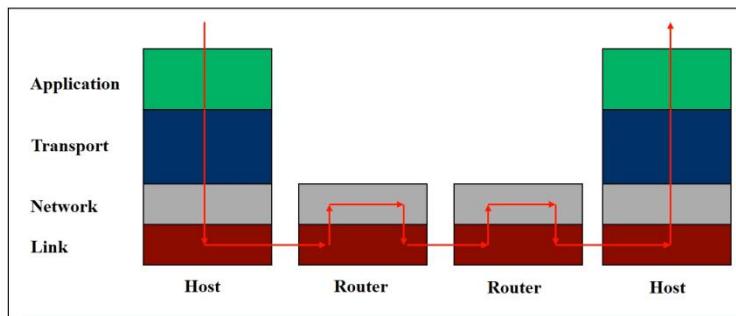
### 7. Allow resource accountability

20

## IP Layering (Principle 2)



- Relatively simple
- Sometimes taken too far



第二个就是要简单化，每一层只做一个事情。

## Goal 1: Survivability



- If network is disrupted and reconfigured...
  - Communicating entities should not care!
  - No higher-level state reconfiguration
- How to achieve such reliability?
  - Where can communication state be stored?

	Network	Host
Failure handing	Replication	“Fate sharing”
Net Engineering	Tough	Simple
Switches	Maintain state	Stateless
Host trust	Less	More

**2022/2/22**

学完物理层就知道，它做到了可以传 0101 过来了。不同的网络传输机制有不同的传输属性，如延迟、吞吐等。光纤的 0101 无非来的快一点，而卫星就来的慢一些。有了 0101 以后，这些数据还不是数据包。物理层的特性无非是 RTT, throughput (1G / 10G) 等。

## Data Link Layer Design Issues

- Services Provided to the Network Layer
- Framing
- Error Control
- Flow Control

有了 0101 以后，我们需要把它存成块，也就是我们需要知道一段数据从哪开始到哪结束，也就是要有一个个数据包的概念。一般 mac 层把数据包叫做 Frame，到了 IP/TCP 层就叫为 package。QoS 这种功能一般是 TCP 层提供的。Mac 层和 IP 层耦合度很高，这也是为什么 7 层变成了 5 层。

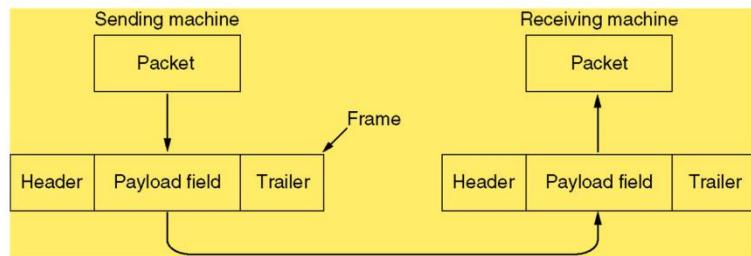
传过来的数据，我们需要存成一块向网络层提供服务。比如说，如果 0101 传输的时候 01 传错了该怎么办，我们需要改正错误。我们在以太网里，一旦有了错误，更多的是把包扔掉，扔掉了以后，流就会缺少数据，比如一张图片少了一块没有办法正确打开，checksum 失败。

## Functions of the Data Link Layer

- Provide service interface to the network layer
- Dealing with transmission errors
- Regulating data flow
- Slow receivers not swamped by fast senders
- Feedback-based flow control
- Rate-based flow control

所以，数据连接层要往网络层提供 IP 包，承担了物理层到传输层的过渡作用。我们需要做 **Regulation**，把数据流变得平滑一点。网络至少也是一个发送、一个接收，发送端太快了也不行，因为接收端来不及接收，所以它们的配合也是很重要的。然后我们也需要有一个 **flow control** 的环节，比如误码纠错的情况。

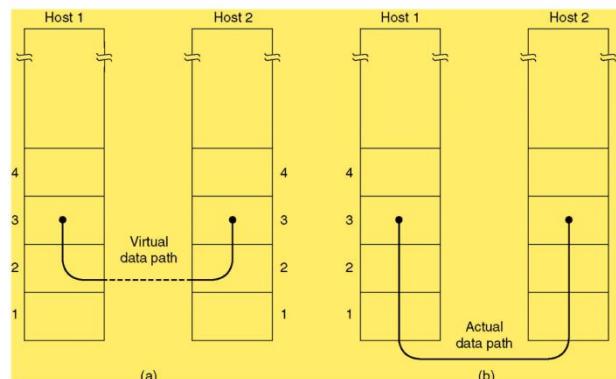
### Break Data Stream into Frames



Relationship between packets and frames.

现在我们看到的就是物理层往上给的一堆不间断的 010101。我们先要知道从哪开始从哪结束，我们需要有标志位来标志一下。

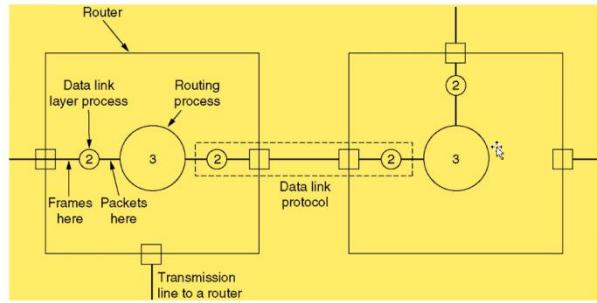
### Services Provided to Network Layer



(a) Virtual communication.  
(b) Actual communication.

我们现在有物理层的支持，底层就可以不管了。

## Services Provided to Network Layer (2)

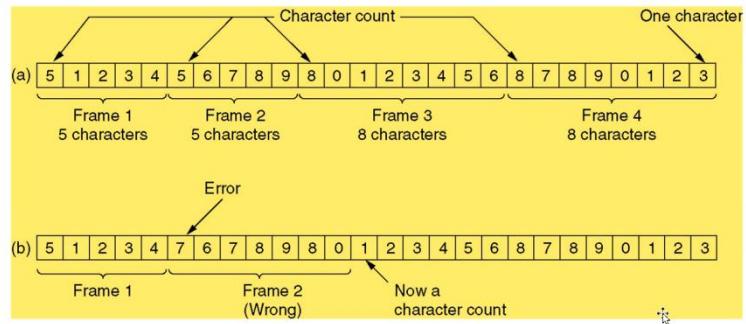


Placement of the data link protocol.

这是交换机 (switch)，二层是可以做交换的。

### 通过计数方式来分包

#### Framing using Character Count



A character stream.

(a) Without errors.

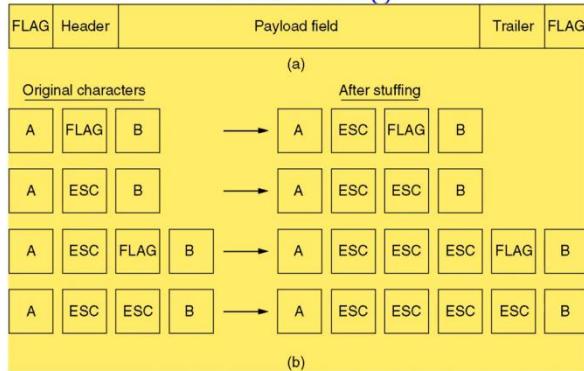
(b)

With one error.

有了 0101 之后，就可以有更高的抽象。我们需要知道头尾。第一个方法就是比较简单粗暴不实用的方法。也就是每个数据包的第一个字节来标志我们要发送几个字节。

这个方法的问题就是，一旦有错误，错误就会被蔓延到后面的所有包，永远不能恢复了。自此开始，数据全乱了。

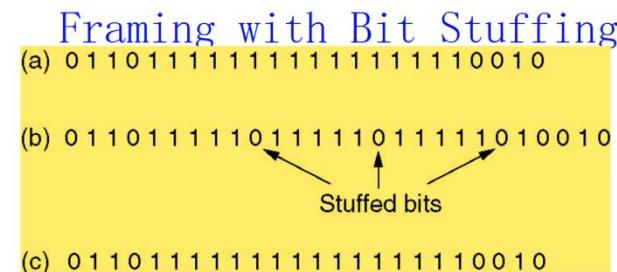
## Framing Using Flag Bytes - Byte Stuffing



我们放一个 Flag (固定串的 0101)，比如用 96 位 bit (如连续 96 个 0) 来表示这个 Flag。这个网络带宽的浪费是很严重的。但是在数据包中，96 个 0 也是很容易产生的。上层的应用总是可能产生和 Flag 序列相同的数据包。数据串里如果有和 Flag 相同的，那就加一个转义符 ESC，同样地，转义符也是由一串 0 和 1 组成的，转义完也有可能继续和 ESC FLAG 相同，那么我们必须继续再加转义符。

所以，ESC ESC 我们会当做 ESC 来用，ESC FLAG 我们也当做 FLAG 来用。这些功能基本上卸载到芯片中了，芯片会记忆这些特殊的字符串。所以上层的工程师在做编码的时候，也可能避免使用这样的用法，比如编码和 FLAG 冲突很高的情况就会加很多转义符。

在数据里，可以采用优化一点的编码方式：



Bit stuffing with starting and ending flags

(0111 1110)

- (a) The original data.
- (b) The data as they appear on the line (stuff a “0” after five consecutive “1”s).
- (c) The data as they are stored in receiver’s memory after de-stuffing.

我们可以在应用层处理掉，比如 FLAG=01111110 的情况，我们可以在应用层对连续的 5 个 1 后面加一个 0 的情况。这是多媒体很标准的编码，它就可以避免和 FLAG 冲突的情况。这个课程讲的更多的是示意，而并非真实情况。

到目前为止，形成一帧这件事情就解决了，不同的物理层的形成一帧的方式是不一样的，比如以太网中是 96 个 0 作为 FLAG，而 wifi / 卫星中可能不一样。

# 错误识别和改正

## Error Detection and Correction

- Error-Correcting Codes
  - ◆ Enough code for receivers to know exactly what have been sent.
- Error-Detecting Codes
  - ◆ Enough code for receivers to know if error has occurred.

接下来我们就要讲检测物理层中的错误和改正的情况。我们已经找到了头和尾，但是我们怎么发现其中有 0 和 1 发生了翻转呢？由于二进制，我们知道一个地方错了就可以反转那一位来改正。

## Hamming Distance

- Number of bit positions that two code words differ, e.g.,

1000 1001

1011 0001

0011 1000

- To detect  $d$  errors, we just need a distance  $d+1$  code
- E.g., **0000 0000 00**, **0000 0111 11**, **1111 1000 00**, and **1111 1111 11**
- The above code has distance 5 and can correct double errors
- $m$  message bits,  $r$  check bits
- $2^m$  legal message, each has  $n$  illegal distance 1 codewords
- $(n+1)2^m \leq 2^n$ ;  $n = r+m$  thus  $(m+r+1) \leq 2^r$

汉明距离的作用就是一个 10 位的，就可以识别出 2 个错误。我们要保证，在汉明距离一下错误的 Bit 的最大数量是小于 2 的。

## Compute Hamming Codes

- Each parity bit calculates the parity for some of the bits in the code word.
- General rule for position  $n$ : skip  $n-1$  bits, check  $n$  bits, skip  $n$  bits, check  $n$  bits, e.g.,
  - Position 1 ( $n=1$ ): skip 0 bit ( $0=n-1$ ), check 1 bit ( $n$ ), skip 1 bit ( $n$ ), check 1 bit ( $n$ ), skip 1 bit ( $n$ ), etc.
  - Position 2 ( $n=2$ ): skip 1 bit ( $1=n-1$ ), check 2 bits ( $n$ ), skip 2 bits ( $n$ ), check 2 bits ( $n$ ), skip 2 bits ( $n$ ), etc.
  - Position 4 ( $n=4$ ): skip 3 bits ( $3=n-1$ ), check 4 bits ( $n$ ), skip 4 bits ( $n$ ), check 4 bits ( $n$ ), skip 4 bits ( $n$ ), etc.

图 1 汉明距离的编码方式

找到位置做奇偶校验。

## Encode Example

	$p_1$	$p_2$	$d_1$	$p_3$	$d_2$	$d_3$	$d_4$	$p_4$	$d_5$	$d_6$	$d_7$
Data word (without parity):			0		1	1	0		1	0	1
$p_1$			1		0	1		0	1	1	
$p_2$			0	0			1	0		0	1
$p_3$					0	1	1	0			
$p_4$								0	1	0	1
Data word (with parity):	1	0	0	0	1	1	0	0	1	0	1

- Data word: 0110 101
- Encoded word: 1000 1100 101

因为是针对流式数据的，所以可以根据流式数据不停地做。

## Error Correction Example

	$p_1$	$p_2$	$d_1$	$p_3$	$d_2$	$d_3$	$d_4$	$p_4$	$d_5$	$d_6$	$d_7$	Parity check	Parity bit
Received data word:	1	0	0	0	1	1	0	0	1	0	0		
$p_1$	1	0	1	0	0	1	0	0	Fail		1		
$p_2$	0	0		1	0		0	0	Fail		1		
$p_3$			0	1	1	0			Pass		0		
$p_4$					0	1	0	0	Fail		1		

- 1000 1100 101 changed to 1000 1100 100
- Flipped bit is: 1011, i.e.,

$$\begin{array}{c}
 p_4 \ p_3 \ p_2 \ p_1 \\
 \text{Binary } 1 \ 0 \ 1 \ 1 \\
 \text{Decimal } 8 \quad 2 \ 1 \ \Sigma = 11
 \end{array}$$

我们隔一位 check 一下。但是这个效率还是不高。

## Error-Correcting Codes

Char.	ASCII	Check bits
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	01111001111
o	0100000	10011000000
c	1100011	11111000011
o	1101111	10101011111
d	1100100	11111001100
e	1100101	00111000101

Order of bit transmission

Use of a Hamming code to correct burst errors.

- If  $\text{burst} < k$  bits, at most one single bit error in each codeword.
- $k$  checkbits correct  $km$  message with burst of  $k$  error.

我们可以做一个整体的打包。

## Why Error Detection?

- Error correction code has high overhead
  - Error rate is  $10^{-6}$  per bit; block size is 1000 bits; 1000,000 bits transferred
  - Error correction, 10 bits are needed per block
    - Total 10,000 extra bits
  - Error detection, 1 parity bit per block suffices
    - Total:  $1000 + 1001$  (retransmit) = 2001 extra bits

有了 error correction, 为什么还需要 error detection? 假设错误概率是 $10^{-6}$ , 每个 block 是 1000 个 bit, 为了做 error correction, 我们需要传 10000 个额外的 bit, 而做 error detection, 我们只需要传 2001 个额外的 bit。

## CRC (循环冗余校验)

### Cyclic Redundancy Check (CRC)

- Also known as polynomial code
  - A bit string representing polynomials with coefficients of 0s and 1s
  - E.g.,  $110001$  represents  $x^5 + x^4 + x^0$
- Sender & receiver agree upon a generator polynomial  $G(x)$ 
  - Let  $r$  be the degree of  $G(x)$ . Append  $r$  zero bits to frame:  $m+r$  bits  $x^r M(x)$
  - Divide the  $m+r$  bits by  $G(x)$ 
    - Subtract the remainder from bit string  $x^r M(x)$ . Call it  $T(x)$
    - Transmit  $T(x)$  to receiver.
- Receiver gets  $T(x) + E(x)$ 
  - $(T(x) + E(x))/G(x) = 0 + E(x)/G(x) = E(x)/G(x)$

CRC 就是一串码代表一个多项式, 然后做一个除法, 如果等于零, 那么包就没错; 如果有余数, 那么就说明有错, 但是错哪儿不知道。

## Calculate the Error-Detecting Codes

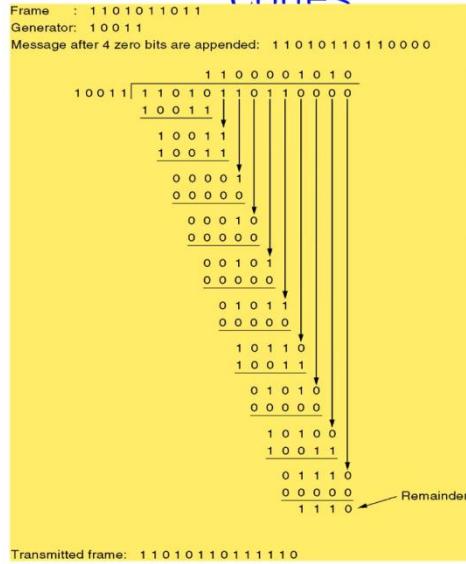


图 2 CRC 除法的例子

如果有余数，就说明信息有错了。因为发送端是把 CRC 乘到信息里的，那么接收端除一下应该是没有余数的。

## CRC – Example Decoding (No Errors)

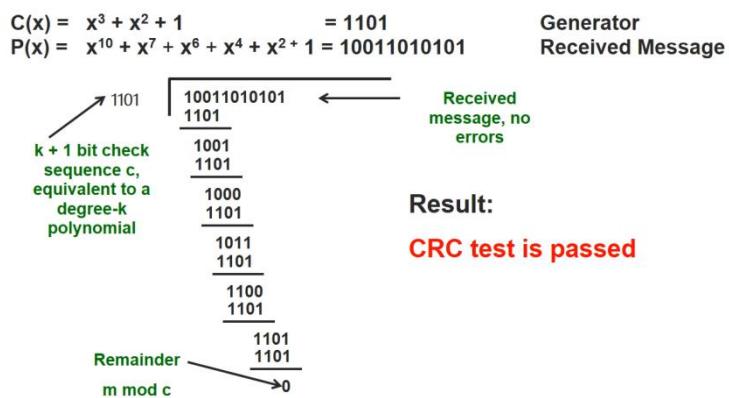


图 3 CRC 没有错误的情况

数据包的前几个 bit 一定会告诉你 CRC 的 version，一定要保证发送端和接收端使用的是同一个 CRC 版本。

## Power of CRC

- Receiver gets  $T(x) + E(x)$ , we are computing  $E(x)/G(x)$
- If single bit error,  $E(x) = x^j$ , as long as  $G(x)$  contains two or more terms
  - If two bit error,  $E(x) = x^i + x^j = x^j(x^{i-j} + 1)$
  - $G(x)$  does not divide  $x^k + 1$  for any  $k$  up to the maximum value of  $i-j$
  - $x^{15} + x^{14} + 1$  will not divide  $x^k + 1$  for any value of  $k$  below 32,768
  - Odd number of bits in error: making  $x+1$  a factor of  $G(x)$
  - Contradiction:  $E(x) = (x+1) Q(x)$ , let  $x=1$
  - A polynomial code with  $r$  check bits will detect all burst errors of length  $\leq r$

CRC 只能告诉你里面有错，需要重传。

## Error Detection vs. Error Correction

- Detection
  - Pro: Overhead only on messages with errors
  - Con: Cost in bandwidth and latency for retransmissions
- Correction
  - Pro: Quick recovery
  - Con: Overhead on all messages
- What should we use?
  - Correction if retransmission is too expensive
  - Correction if probability of errors is high

Correction 有计算量还需要有额外的开销。

## Mac 层协议

前面我们也讲过了，物理层的不可靠可能是一个 bit 在信号处理的时候搞错了。更多是路由器处理不过来，buffer overflow 直接扔掉了。我们物理层出错在传到第三层之前就会做 checksum，所以对于发送端，我们是不知道最终接收到了错误的数据包还是在中间丢掉了。那么我们开始看一下网络的代码。

# Protocol Definitions

```
#define MAX_PKT 1024      /* maximum packet size */

typedef enum {false, true} boolean;
typedef unsigned int seq_nr;    /* sequence or ack numbers */
typedef struct {
    unsigned char data[MAX_PKT];
} packet;                      /* packet definition */
typedef enum {data, ack, nak} frame_kind;

typedef struct {
    frame_kind kind;        /* what kind of frame */
    seq_nr seq;             /* sequence number */
    seq_nr ack;             /* acknowledgement number */
    packet info;            /* the network layer packet */
} frame;
```

我们的 TCP/IP 是 1500 个 Byte。有一个 int 的 sequence\_number，我们需要对数据包有一个编号。帧数据的类型也有 data, ack, nak。发送端要有一个发送的序列号，以及接收端可以有一个接收的序列号。

## Protocol Definition (ctd.)

```
#define MAX_PKT 1024          /* determines packet size in bytes */
typedef enum {false, true} boolean;           /* boolean type */
typedef unsigned int seq_nr;                 /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet; /* packet definition */
typedef enum {data, ack, nak} frame_kind;     /* frame kind definition */

typedef struct {                                /* frames are transported in this layer */
    frame_kind kind;                          /* what kind of a frame is it? */
    seq_nr seq;                            /* sequence number */
    seq_nr ack;                           /* acknowledgement number */
    packet info;                         /* the network layer packet */
} frame;

/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void get_packet_from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

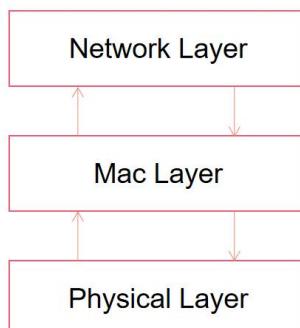
/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc(k) is expanded in-line: Increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

其实 API 很简单，有一个 to 和 from。物理层一对，网络层一对，我们目前在讲 mac 层（数据链接层）怎么做的，如下图所示：



还有一些 timer 函数，比如发了一个包以后，多久以后还没人告诉我收到了，那么我们就认为这个包丢了。Enable/disable 就是告诉网络层要不要继续传输数据过来了。event 是物理层产生的中断，如网卡有一个数据包过来时触发的中断。

## 无限制的单工协议

- 数据单向传送
- 收发双方的网络层都处于就绪状态（随时待命）
- 处理时间忽略不计（瞬间完成）
- 可用的缓存空间无穷大（无限空间）

```
1.  typedef enum {frame_arrival} event_type;
2.  #include"protocol.h"
3.  void sender1(void) {
4.      frame s;
5.      packet buffer;
6.      while(true) {
7.          from_network_layer(&buffer);           //从网络层获取数据
8.          s.info = buffer;
9.          to_physical_layer(&s);                //传输到物理层
10.     }
11. }
12.
13. void receiver1(void) {
14.     frame r;
15.     event_type event;
16.     while(true) {
17.         wait_for_event(&event);              //等到某个事件发生
18.         from_physical_layer(&r);           //从物理层获取 frame
19.         to_network_layer(&r.info);        //将数据传输到网络层
20.     }
21. }
```

这里的 sender 就是弄了一个 frame 和 packet，做死循环：从网络层拿一个数据包，然后封装成一个 frame（添加 mac 层头包尾），然后往物理层一发。

而 receiver 是要等待物理层的事件，然后把数据包拿出来往上传递。wait\_for\_event 这个函数的消耗非常高，因为物理层一个中断上来，需要上下文切换到拿包的线程。这个处理就需要耗费 CPU 几百个 cycle，然后再交付给上层。针对我们几十 G 的网络带宽，需要 4 个 CPU。

但是这个协议比较 naive，完全听天由命。两边没有任何同步的事情。

## 单工的 stop-and-wait 协议

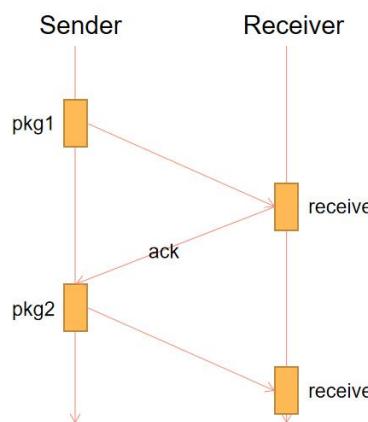
这个协议可以避免接收方被发送方的数据淹没。

```

1.  typedef enum {frame_arrival} event_type;
2.  #include"protocol.h"
3.  void sender2(void) {
4.      frame s;
5.      packet buffer;
6.      event_type event;
7.      while(true) {
8.          from_network_layer(&buffer);           //从网络层获取数据
9.          s.info = buffer;
10.         to_physical_layer(&s);              //传输到物理层
11.         wait_for_event(&event);            //不再继续进行直到收到 ack 事件
12.     }
13. }
14. void receiver2(void) {
15.     frame r, s;
16.     event_type event;
17.     while(true)
18.     {
19.         wait_for_event(&event);          //等到数据包到达
20.         from_physical_layer(&r);       //从物理层获取 frame
21.         to_network_layer(&r.info);    //将数据传输到网络层
22.         to_physical_layer(&s);        //向对方发送一个 ack
23.     }
24. }

```

`stop-and-wait` 是这个协议的高度的总结。我们要等待物理层给我们信息过来，它们的协作过程和之前差不多，但是加了一个同步。`sender` 发了一个 package 给 `receiver` 以后，就停在那里了。`receiver` 收到以后会给 `sender` 发一个 ack 帧，`sender` 收到 ack 以后再继续发下一个。



`sender` 发送完以后阻塞在 `wait_for_event` 上等待 `receiver` 发送 `ack` 回来。但是这个协议的问题就是 `sender` 阻塞的这段时间什么都没干。

## 有错误信道的单工协议

信道不再是完美信道：有噪声就会产生差错，有差错就可能会引起以下这些问题

Q1：有可能收到重复帧，如何解决？

A1：给每个帧一个独一无二的序列号，序号也可以用来重组排序

Q2：收方接到了帧，但发现这个帧出现了错误通不过校验，怎么解决？

A2：增加对正确帧的确认，只有收到的帧通过校验，才向 `sender` 发送一个确认。

Q3：任何的帧（包括确认帧）都会在路上丢失，这样会导致 `receiver` 一直等下去怎么办？

A3：肯定确认重传(PAR)（或称自动重传请求(ARQ)）技术：发送方启动一个重传定时器，超时前如果收到收方的确认，拆除定时器，超时还未收到确认，重传并重置定时器。

```
1.  typedef enum {frame_arrival} event_type;
2.  #include"protocol.h"
3.  void sender3(void) {
4.      seq_nr next_frame_to_send;           // 下一个即将出去的 frame 的 sequence number
5.      frame s;
6.      packet buffer;
7.      event_type event;
8.      next_frame_to_send = 0;
9.      from_network_layer(&buffer);
10.     while(true) {
11.         s.info = buffer;
12.         s.seq = next_frame_to_send;
13.         to_physical_layer(&s);
14.         start_timer(s.seq);             // 设置定时器
15.         wait_for_event(&event);
16.         if(event == frame_arrival) {    // 收到了回复帧
17.             from_physical_layer(&s);    // 从物理层里面拿数据
18.             if(s.ack == next_frame_to_send) { // ack 等于 next_frame_to_send 意味着当前这个包已经收到了。
19.                 stop_timer(s.ack);        // 关闭定时器
20.                 from_network_layer(&buffer);
21.                 inc(next_frame_to_send); // 准备传递下一帧
22.             }
23.         }
24.         else {                         // 否则可能是定时器超时，重新发送这一帧
25.         }
26.     }
27. }
28. void receiver3(void) {
29.     seq_nr frame_expected;
30.     frame r, s;
31.     event_type event;
```

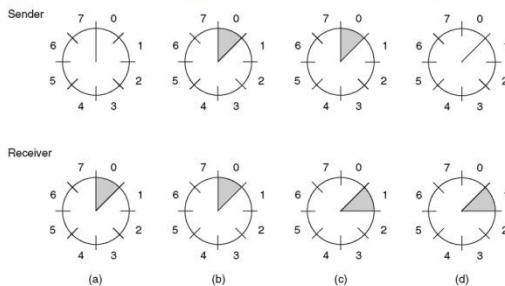
```

32.     frame_expected = 0;
33.     while(true) {
34.         wait_for_event(&event);
35.         if(event == frame_arrival) {
36.             from_physical_layer(&r);
37.             if(r.seq == frame_expected) { // 接收的的确是我们在等待的
38.                 to_network_layer(&r.info);
39.                 inc(frame_expected); //继续等下一帧
40.             }
41.             s.ack = 1 - frame_expected; //对收到的帧进行确认
42.             to_physical_layer(&s); //传回发送方
43.         }
44.     }
45. }
```

我们打算在信息里装 ACK，而不是等待中断。我们现在有了 seq\_number 的概念。如果我们收到一个 event == frame\_arrival 事件，我们把数据包拿回来，如果 ack == next\_frame\_to\_send，我们就终止掉 timer，并且开始发下一个包，这样我们长时间没收到 ack 就不会出现 sender 被锁死的情况。

receiver 来说，如果是 frame\_expected，那么就是 1 已经打开了，1 其实还没发呢。

## Simple Example of Sliding Window



A sliding window of size 1, with a 3-bit sequence number.

- (a) Initially.
- (b) After the first frame has been sent.
- (c) After the first frame has been received.
- (d) After the first acknowledgment has been received.

收到以后，receiver 会给 sender 发送 ack，告诉它 receiver 已经准备开始收第二个包了。

# Sliding Window Protocols

## •Observations

- Duplex communication
- Data packet can piggyback acks

## •Protocols

- A One-Bit Sliding Window Protocol
- A Protocol Using Go Back N
- A Protocol Using Selective Repeat

## 滑动窗口协议

我们讲网络的时候没有发送和接收的区别，必须同时能发和收，所以协议栈不能两边不一样。所以我们是双工的通信：既能发又能收。

```
1. // 滑动窗口协议是双向的
2. #define MAX_SEQ 1
3. typedef enum {frame_arrival, cksum_err, timeout} event_type;
4. #include "protocol.h"
5. void protocol4 (void) {
6.     seq_nr next_frame_to_send;          // 0 或 1
7.     seq_nr frame_expected;           // 0 或 1
8.     frame r, s;
9.     packet buffer;
10.    event_type event;
11.    next_frame_to_send = 0;           // 输出流的下一帧的 frame
12.    frame_expected = 0;             // 期待接收的帧号
13.    from_network_layer(&buffer);
14.    s.info = buffer;
15.    s.seq = next_frame_to_send;
16.    s.ack = 1 - frame_expected;      // 附带响应
17.    to_physical_layer(&s);
18.    start_timer(s.seq);
19.    while (true) {
20.        wait_for_event(&event);
21.        if (event == frame_arrival) {
22.            from_physical_layer(&r);           // 从物理层拿到帧
23.            if (r.seq == frame_expected) {      // 到达的包确实是按照顺序的包 (receiver 的行为)
24.                to_network_layer(&r.info);
25.                inc(frame_expected);
26.            }
27.            if (r.ack == next_frame_to_send) { // 到达的包是 receiver 发给 sender 的 ack
28.                stop_timer(r.ack);           // 停止计时器

```

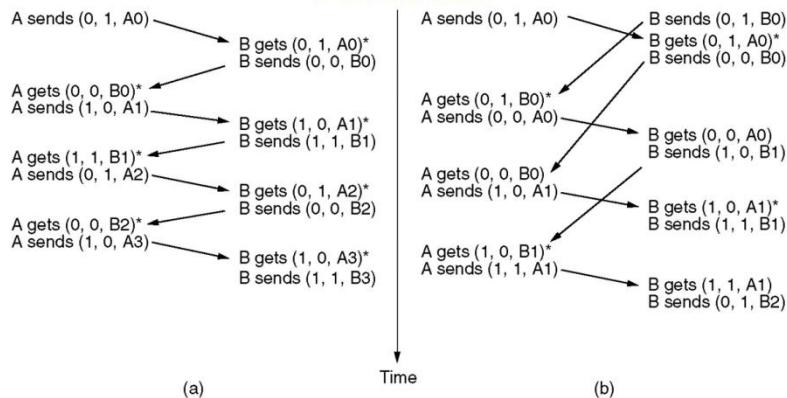
```

29.         from_network_layer(&buffer);
30.         inc(next_frame_to_send);
31.     }
32. }
33. s.info = buffer;
34. s.seq = next_frame_to_send;
35. s.ack = 1 - frame_expected; // 最后收到的帧的 seq number
36. to_physical_layer(&s);
37. start_timer(s.seq); // 设置定时器
38. }
39. }

```

protocol 就是既能发又能收，没有 sender 和 receiver 的区别了。等待 event，如果是 next\_frame\_to\_send，我们就要发送下一个包；如果是收到的数据包，那么就是把物理层收到的数据包交给网络层。但是这里面还是有很多问题的：

## A peculiarity for One-Bit Sliding Window Protocol



(a) Normal case.

(b) Abnormal case: half frames contain duplicates

Notation is (*seq, ack, packet number*).

\* indicates the network layer accepts a packet.

但是又出来新的问题了，(a)是正常的情况。但是如果两个同时发的情况下，B 已经发出去了，B 就认为发出去的 B0 没收到。所以这是两个第一个数据包没有做协同导致的问题。所以理论上每个数据都要发多次才行。那么我们的 socket 怎么解决这个问题呢？

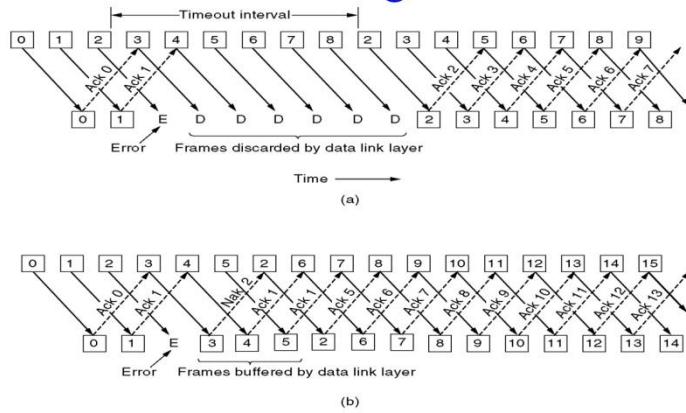
socket 中有 client 和 server 的概念，server 等待 client 的请求，只有 client 发送请求了以后两个人才开始通信，就避免了这种情况。

## Bandwidth Utilization for 1-bit Protocol

- E.g.,
  - 50-kbps satellite channel
  - 500 ms round-trip delay
  - Assume 20 ms for sending a frame
  - Acknowledgement can return as early as 520 ms later
  - i.e.,  $500/520=96\%$  unused channel

round-trip 时间很长，500ms，假设我们的一个数据包要 20ms。那么中间这个就需要 500ms，那么有 96% 的时间和带宽都浪费掉了。所以我们选用 no-blocking 的机制。

### A Protocol Using Go Back N



Pipelining and error recovery. Effect on an error when  
(a) Receiver's window size is 1.  
(b) Receiver's window size is large.

那么我们就需要有 sequence-number 的机制。我们发送数据包在第三个版本已经加入了 timeout 的情况了，0、1 收到了 2、3、4、5、6 继续往前发，但是 2 没有收到。这样 2 就会 timeout，发送 nak 后等待 2 继续发包。然后我们已经收到的 3、4、5 可以不用丢掉。

## 2022/2/25 (NO)

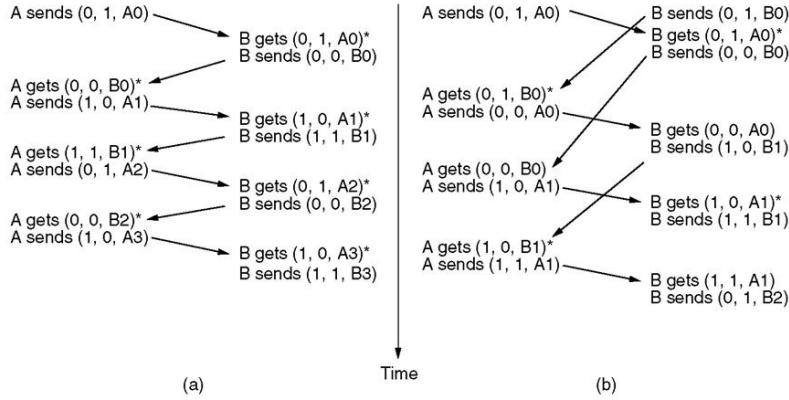
CRC 的能力不是无穷的，它里面是一个值乘以一个数，接收端再除掉变成零。但是错误的几位有可能正好被整除，所以所有东西都不是永远正确的。CRC 有不同的版本，有不同的长短。CRC 一般是 32 位以下的。checksum 会在各个层都做，因为上一层会不信任底层的 checksum。越往上层走越复杂。

形成一帧怎么做，我们回顾一下这节课。以太网就是 96 个 0，FLAG 会和信息重合，就需要转义符、转义符的转义符等。网络中错误的概率一定不等于 0，只是说加上了一个转义符以后，错误的概率会下降。以太网的错误的概率就是  $10^{-5}$ 。汉明编码必须在汉明距离之内才能进行纠错。如果数字特别长的情况下，在高位错一个 1，可能就没有办法被 CRC 纠错出来。因为 MAC 层在物理层之上，还有一个 top-down 的角度会有一些进展。所以，我们需要知道纠错、检测错误还是有能力上限的。然后我们又讨论了一下纠错和检错各自的优缺点

点，它们有不同的适用场景不能同时使用。

我们一开始讲了 6 个版本的协议，前三个版本都有 sender、receiver 的角色。我们讲了中断的 event 一下子就需要几百个 cycle。一对对的 to/from，这里面不能进行数据 copy，这是一个非常大的系统开销。所以 DPDK 中有 zero-copy，也就是使用指针做 reference。第四个版本把 sender/receiver 糊在一起了，第五第六个版本就是 sliding window 的概念。

## A peculiarity for One-Bit Sliding Window Protocol



(a) Normal case.

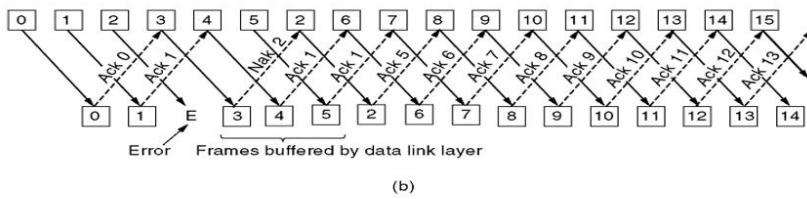
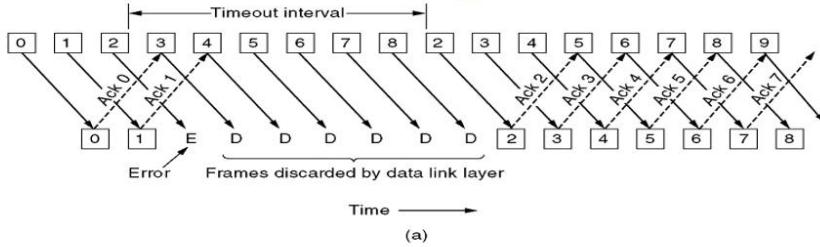
(b) Abnormal case: half frames contain duplicates

Notation is (*seq, ack, packet number*).

\* indicates the network layer accepts a packet.

极端情况下所有的数据都会传两遍。现在新型的 server 机制就是单线程的，处理完了再处理下一个。

## A Protocol Using Go Back N



Pipelining and error recovery. Effect on an error when

(a) Receiver's window size is 1.

(b) Receiver's window size is large.

window size = 1 的(a)图只能存一个，如果不是我们要的那个，就需要扔掉。所以代码要求 packet 过来要按照顺序的。如果我们把接收到的数据包 buffer 起来，就不需要扔掉了。Select Repeat 就是类似于(b)之类的机制。一旦 2 到了以后，我们就直接挪过 3,4,5 开始准备

接收 6。

## 多路接收协议

```
78/* Protocol 5 (pipelining) allows multiple outstanding frames. The sender
1.      * may transmit up to MAX_SEQ frames without waiting for an ack. In addition,
2.      * unlike the previous protocols, the network layer is not assumed to have
3.      * a new packet all the time. Instead, the network layer causes a
4.      * network_layer_ready event when there is a packet to send.
5.      */
6.
7. #define MAX_SEQ 7      /* should be 2^n - 1 */
8. typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready} event_type;
9. #include "protocol.h"
10.
11. static boolean between(seq_nr a, seq_nr b, seq_nr c)
12. {
13.     /* Return true if (a <= b < c circularly; false otherwise. */
14.     if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
15.         return(true);
16.     else
17.         return(false);
18. }
19.
20. static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
21. {
22.     /* Construct and send a data frame. */
23.     frame s;      /* scratch variable */
24.
25.     init_frame(&s);
26.     s.kind = data;
27.     s.info = buffer[frame_nr];    /* insert packet into frame */
28.     s.seq = frame_nr;    /* insert sequence number into frame */
29.     s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);    /* piggyback ack */
30.     to_physical_layer(&s);    /* transmit the frame */
31.     start_timer(frame_nr);    /* start the timer running */
32. }
33.
34. void protocol5(void)
35. {
36.     seq_nr next_frame_to_send;    /* MAX_SEQ > 1; used for outbound stream */
37.     seq_nr ack_expected; /* oldest frame as yet unacknowledged */
38.     seq_nr frame_expected; /* next frame expected on inbound stream */
```

```

39.     frame r;      /* scratch variable */
40.     packet buffer[MAX_SEQ+1];    /* buffers for the outbound stream */
41.     seq_nr nbuffered;    /* # output buffers currently in use */
42.     seq_nr i;      /* used to index into the buffer array */
43.     event_type event;
44.
45.     enable_network_layer();      /* allow network_layer_ready events */
46.     ack_expected = 0;    /* next ack expected inbound */
47.     next_frame_to_send = 0;    /* next frame going out */
48.     frame_expected = 0;    /* number of frame expected inbound */
49.     nbuffered = 0;      /* initially no packets are buffered */
50.
51.     while (true) {
52.         wait_for_event(&event);    /* four possibilities: see event_type above */
53.
54.         switch(event) {
55.             case network_layer_ready:    /* the network layer has a packet to send */
56.                 /* Accept, save, and transmit a new frame. */
57.                 from_network_layer(&buffer[next_frame_to_send]); /* fetch new packet */
58.                 nbuffered = nbuffered + 1;    /* expand the sender's window */
59.                 send_data(next_frame_to_send, frame_expected, buffer); /* transmit the frame */
60.                 inc(next_frame_to_send);    /* advance sender's upper window edge */
61.                 break;
62.
63.             case frame_arrival:    /* a data or control frame has arrived */
64.                 from_physical_layer(&r);    /* get incoming frame from physical layer */
65.
66.                 if (r.seq == frame_expected) {
67.                     /* Frames are accepted only in order. */
68.                     to_network_layer(&r.info); /* pass packet to network layer */
69.                     inc(frame_expected);    /* advance lower edge of receiver's window */
70.                 }
71.
72.                 /* Ack n implies n - 1, n - 2, etc. Check for this. */
73.                 while (between(ack_expected, r.ack, next_frame_to_send)) {
74.                     /* Handle piggybacked ack. */
75.                     nbuffered = nbuffered - 1;    /* one frame fewer buffered */
76.                     stop_timer(ack_expected);    /* frame arrived intact; stop timer */
77.                     inc(ack_expected);    /* contract sender's window */
78.                 }
79.                 break;
80.
81.             case cksum_err: ;    /* just ignore bad frames */
82.                 break;

```

```

83.
84.     case timeout: /* trouble; retransmit all outstanding frames */
85.         next_frame_to_send = ack_expected; /* start retransmitting here */
86.         for (i = 1; i <= nbuffered; i++) {
87.             send_data(next_frame_to_send, frame_expected, buffer); /* resend 1 frame */
88.             inc(next_frame_to_send); /* prepare to send the next one */
89.         }
90.     }
91.
92.     if (nbuffered < MAX_SEQ)
93.         enable_network_layer();
94.     else
95.         disable_network_layer();
96.     }
97. }
```

首先它有一个 `bitmap` 标志着对应编号的数据来了没有，然后还有包的 `buffer`。这个代码里，`network_layer_ready` 对应的就是从网络层往下发，这个还是比较容易的

## 非顺序接收协议

```

1.  /* Protocol 6 (nonsequential receive) accepts frames out of order, but
2.   * passes packets to the network layer in order. Associated with each
3.   * outstanding frame is a timer. When the timer goes off, only that frame is
4.   * retransmitted, not all the outstanding frames, as in protocol 5.
5.   *
6.   * To compile: cc -o protocol6 p6.c simulator.o
7.   * To run: protocol6 events timeout pct_loss pct_cksum debug_flags
8.   *
9.   * Written by Andrew S. Tanenbaum
10.  * Revised by Shivakant Mishra
11.  */
12.
13. #define MAX_SEQ 7 /* should be 2^n - 1 */
14. #define NR_BUFS ((MAX_SEQ + 1) / 2)
15. /* changed from MAX_SEQ+1 / 2 ,and back */ /*JH*/
16. typedef enum {frame_arrival, cksum_err, timeout, network_layer_ready, ack_timeout} event_type;
17. #include <unistd.h>
18. #include "protocol.h"
19. boolean no_nak = true; /* no nak has been sent yet */
20.
21. static boolean between(seq_nr a, seq_nr b, seq_nr c)
```

```

22. {
23.     /* Same as between in protocol5, but shorter and more obscure. */
24.     return ((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a));
25. }
26.
27. static void send_frame(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
28. {
29.     /* Construct and send a data, ack, or nak frame. */
30.     frame s;      /* scratch variable */
31.
32.     s.kind = fk;    /* kind == data, ack, or nak */
33.     if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
34.     s.seq = frame_nr;    /* only meaningful for data frames */
35.     s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ + 1);
36.     if (fk == nak) no_nak = false; /* one nak per frame, please */
37.     to_physical_layer(&s); /* transmit the frame */
38.     /* if (fk == data) start_timer(frame_nr % NR_BUFS); */
39.     if (fk == data) start_timer(frame_nr); /*JH*/
40.
41.     stop_ack_timer(); /* no need for separate ack frame */
42. }
43.
44. void protocol6(void)
45. {
46.     seq_nr ack_expected;    /* lower edge of sender's window */
47.     seq_nr next_frame_to_send; /* upper edge of sender's window + 1 */
48.     seq_nr frame_expected; /* lower edge of receiver's window */
49.     seq_nr too_far; /* upper edge of receiver's window + 1 */
50.     int i; /* index into buffer pool */
51.     frame r; /* scratch variable */
52.     packet out_buf[NR_BUFS]; /* buffers for the outbound stream */
53.     packet in_buf[NR_BUFS]; /* buffers for the inbound stream */
54.     boolean arrived[NR_BUFS]; /* inbound bit map */
55.     seq_nr nbuffered; /* how many output buffers currently used */
56.     event_type event;
57.
58.     /* put protocolnumber and process id in logfile */ /*JH*/
59.     sprintf(logbuf,"XXX6 protocol6, pid=%d\n", getpid()); /*JH*/
60.     flog_string(logbuf); /*JH*/
61.
62.     enable_network_layer(); /* initialize */
63.     ack_expected = 0; /* next ack expected on the inbound stream */
64.     next_frame_to_send = 0; /* number of next outgoing frame */
65.     frame_expected = 0; /* frame number expected */

```

```

66.     too_far = NR_BUFS; /* receiver's upper window + 1 */
67.     nbuffered = 0; /* initially no packets are buffered */
68.
69.     for (i = 0; i < NR_BUFS; i++) arrived[i] = false;
70.     while (true) {
71.         wait_for_event(&event); /* five possibilities: see event_type above */
72.         switch(event) {
73.             case network_layer_ready: /* accept, save, and transmit a new frame */
74.                 nbuffered = nbuffered + 1; /* expand the window */
75.                 from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]); /* fetch new packet */
76.
77.                 send_frame(data, next_frame_to_send, frame_expected, out_buf); /* transmit the frame */
78.                 inc(next_frame_to_send); /* advance upper window edge */
79.                 break;
80.             case frame_arrival: /* a data or control frame has arrived */
81.                 from_physical_layer(&r); /* fetch incoming frame from physical layer */
82.                 if (r.kind == data) {
83.                     /* An undamaged frame has arrived. */
84.                     if ((r.seq != frame_expected) && no_nak)
85.                         send_frame(nak, 0, frame_expected, out_buf); else start_ack_timer();
86.
87.                     if (between(frame_expected, r.seq, too_far) && (arrived[r.seq%NR_BUFS] == false)) {
88.                         /* Frames may be accepted in any order. */
89.                         arrived[r.seq % NR_BUFS] = true; /* mark buffer as full */
90.                         in_buf[r.seq % NR_BUFS] = r.info; /* insert data into buffer */
91.                         while (arrived[frame_expected % NR_BUFS]) {
92.                             /* Pass frames and advance window. */
93.                             to_network_layer(&in_buf[frame_expected % NR_BUFS]);
94.
95.                             no_nak = true;
96.                             arrived[frame_expected % NR_BUFS] = false;
97.                             inc(frame_expected); /* advance lower edge of receiver's window */
98.
99.                             inc(too_far); /* advance upper edge of receiver's window */
100.                            start_ack_timer(); /* to see if (a separate ack is needed */
101.                        }
102.                    }
103.                }
104.                if((r.kind==nak) && between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_send)
    )

```

```

105.             send_frame(data, (r.ack+1) % (MAX_SEQ + 1), frame_expected, out_buf);
106.
107.             while (between(ack_expected, r.ack, next_frame_to_send)) {
108.                 nbuffered = nbuffered - 1; /* handle piggybacked ack */
109.                 stop_timer(ack_expected);/*JH*/ /* frame arrived intact */
110.                 inc(ack_expected); /* advance lower edge of sender's window */
111.             }
112.             break;
113.
114.         case cksum_err: if (no_nak) send_frame(nak, 0, frame_expected, out_buf); break; /* damaged frame */
115.         case timeout: send_frame(data, get_timedout_seqnr(), frame_expected, out_buf); break;
116.             /* we timed out */
117.             case ack_timeout: send_frame(ack,0,frame_expected, out_buf); /* ack timer expired;
118.             send ack */
119.         if (nbuffered < NR_BUFS) enable_network_layer(); else disable_network_layer();
120.     }
121. }
```

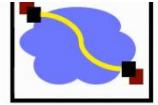
21:22

42: 04

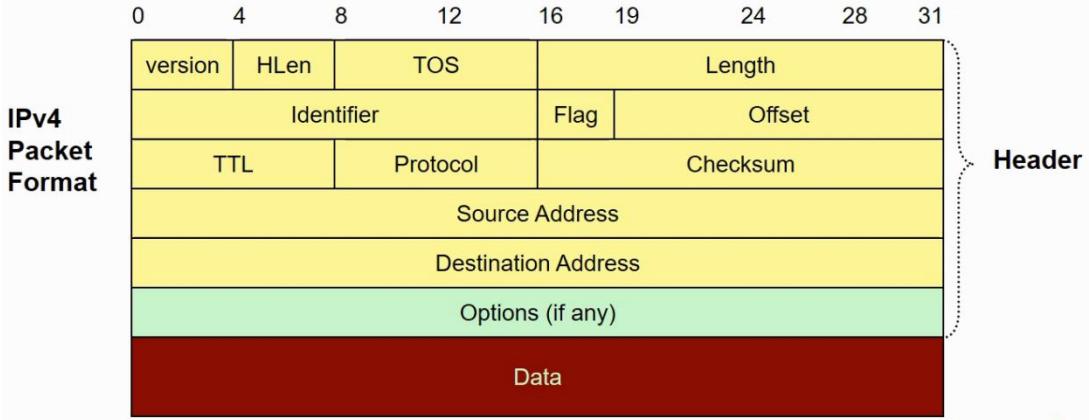
## IP Packets, IPv6 & NAT

传统的 socket 是允许在 kernel 态的，全部在 kernel 里处理完了再还给用户态。但是这里面就会存在一个上下文切换的开销，而 DPDK 是完全运行在用户态的一个网络栈，降低数据包处理中的一个个 overhead，DPDK 还有高效的中断处理、无数据拷贝的数据移动等技巧。我们现在进入 IP 层。

# IP Service Model



- Low-level communication model provided by Internet
- Datagram
  - Each packet self-contained
    - All information needed to get to destination
    - No advance setup or connection maintenance
  - Analogous to letter or telegram

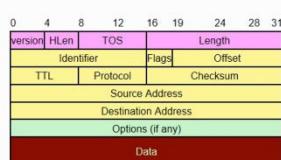


3

这其实就是快递单子怎么贴的问题，发信人地址、收信人地址、发信人手机号、收信人手机号。前面的 version，可能要涉及到同一个版本的 CRC 之类的。然后就是 HLen，就是黄色的部分的长度，因为下面的 Options 也是算在 Header 的长度的。我们通常认为 Header 是 20 个 Byte，认为 Options 是做实验备用的字段。比如 SDN（软件定义交换机）就可以利用包头中新加的几位来 check 一些事情。

第二行就是来做切分网络包的，TTL 就是用来处理网络中的游离包的情况。Protocol 是为了上层协议服务的。

## IPv4 Header Fields



- Version: IP Version
  - 4 for IPv4
- HLen: Header Length
  - 32-bit words (typically 5)
- TOS: Type of Service
  - Priority information
- Length: Packet Length
  - Bytes (including header)
- Header format can change with versions
  - First byte identifies version
- Length field limits packets to 65,535 bytes
  - In practice, break into much smaller packets for network performance considerations
  - How about Jumbo Frame/Packet?

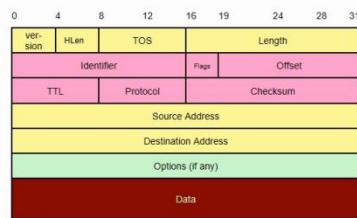
TOS 分为吞吐敏感的还是 throughput 敏感的等。因为 Length 是 16 位，所以 IP 包的流量

上限就是 64K，但是实际上限会比这个还要小。

## IPv4 Header Fields



- Identifier, flags, fragment offset → used primarily for fragmentation
  - Introduce later.
- Time to live
  - Must be decremented at each router
  - Packets with TTL=0 are thrown away
  - Ensure packets exit the network
- Protocol
  - Demultiplexing to higher layer protocols
  - TCP = 6, ICMP = 1, UDP = 17...
  - Rules are no longer kept!
- Header checksum
  - Ensures some degree of header integrity
  - Relatively weak – 16 bit
- Options
  - E.g. Source routing, record route, etc.
  - Performance issues
    - Poorly supported traditionally.



## 2022/3/1 (NO)

## 2022/3/4

Distance Vector 没有全图的概念，只和邻居打交道。我们邻居的邻居可能会带来一些误导信息，比如 infinity 等，这就导致不一定收敛，好处就是不一定收敛。Link State 避免不了广播的信息被篡改，当然我们也可以通过全局信息把网络拓扑结构中错误的信息侦测出来。我们可以对每个点放一个摘要出来，在摘要之间做比较。我们就可以把损坏消息的节点反算出来。

当然这非常小众是领域，我们一般默认没有问题。在 Distance Vector 中的错误比较难弄，比如两个点一直传误导信息导致一直收敛不了。

LS 和 DV 是平铺的，所有节点的地方和交互机制是对等的，这样情况下每个节点都是一个条目。IP 地址 32 位，也就是  $2^{32}$  次方的条目，所以在网络内不会这样。通常是在局域网里才会算 LS 和 DV。

eg：邮递员把信从邮筒里拿出来只负责看是不是这个区的，如果不是上海的，往市局一扔就完了。

所以每个邮递员内存中不要记录这么多，所以我们有一个 least information。所以我们就有个 Areas，它就类似于信件的前几位，所以真正的架构看这个就行了。

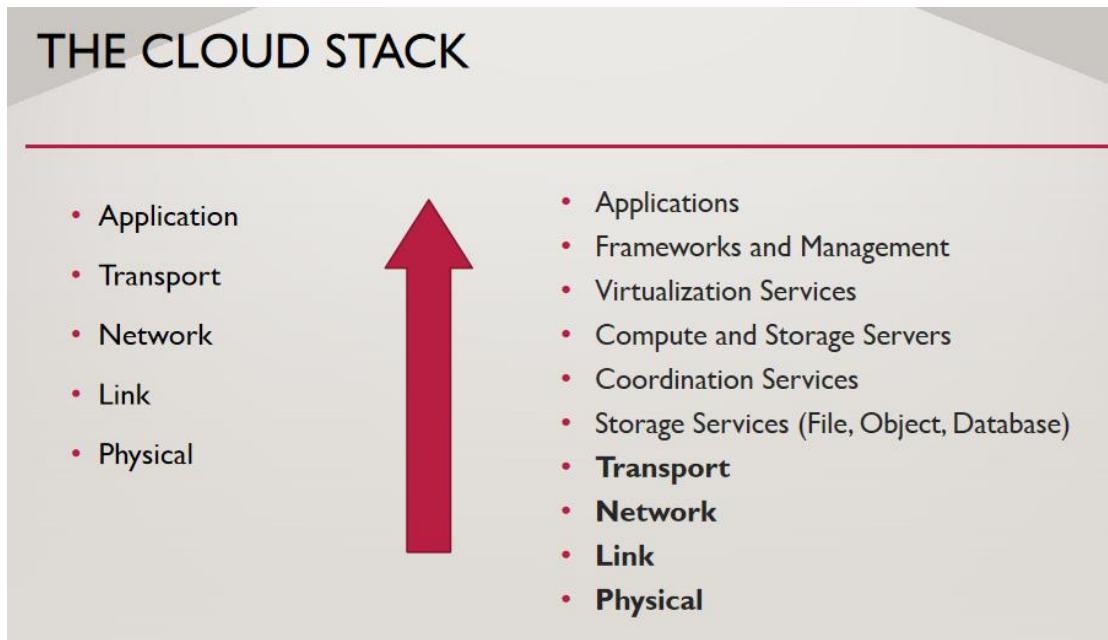
### 《PPT Routing Hierarchy》

我们只需要扔到蓝色的节点就行，再往下扔到哪它也不用管。这几个路由器叫做核心路由器，也会比较贵。核心路由器的路由表也是无穷大的，只需要知道几个子区域的路由器在哪就行了。这就是层次化路由的好处。一个路由器只需要知道往上传还是在本路由里就可以了。

假设这里面有这个用 **single shortest path** 两跳就到了。不一定是跳数最少的是最优路径。所以跳转的时候优先进入大 Area，进来了再说。所以路由不光是最短路径算法。我们讲了 IP 层也讲了隧道技术，很重要。

这个隧道其实可以做很多事情，通俗来讲一个隧道走一趟之后就会变化点东西。明文出来密文也是一种隧道。

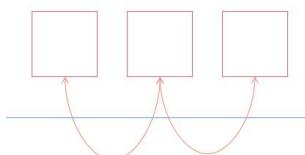
CLOUD STACK 和网络 STACK 既类似又不一样。



我们经常也推广一些新的理念。以前的 OS 是以计算为中心，IO/网络/内存都很慢。不管是什么 OS，最终的目标都是我们的用户。云 OS 是以网络为基点，从下到上来服务应用，剩下的就是细化的问题了。现在的云 OS 还是在之前的扁平化的 OS 中叠加出来一个复杂的协议栈，功能的重复和互相调用的不协调导致云 OS 上多节点的效率非常低。

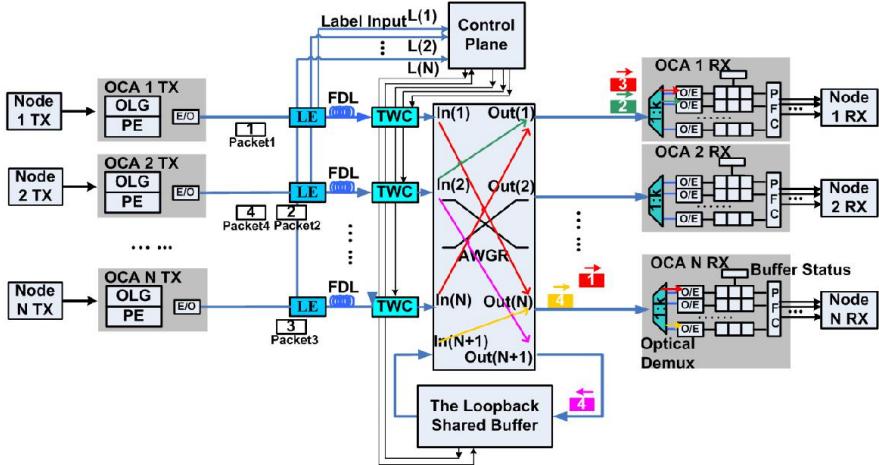
我们今天讲 DataCenter Network 是怎么把资源组织起来的，也就是 IP 在做什么事情。

之前的大圆被认为是不同的数据中心，比如阿里云/华为云的可用区。光速目前还不能突破，在一个 Data Center 中可以忽略光传输的 latency（延时）。DC 的拓扑有很多种，第一种是总线型的，必须有一个仲裁器来避免冲突避免，或者做载道监听等。还有环形的、星型的等。



DC 要把内部结点和 DC 连接在一起，要有效率和鲁棒性，要能迁移和扩容。

Network Switch 只需要大致知道是什么东西，它是 **store and forward**。一个数据包进来以后先存起来，**forward** 就是知道要往哪传。在 Output 口会有调度来判断谁先传谁后传。这个事情可以软件化地做了。



The system diagram of the proposed optical switch. OLG: Optical Label Generator; DE: Delay Element.

左边输入右边输出，数据包暂存在下面的 buffer 中。在右边的 Output 中可以设置输出策略，如 FIFO。下部的 buffer 就叫做 shared memory。crossbar 特点就是简单通用，但是实时性不强，而策略也一定不是 FIFO，比如带 QS 的来保证实时性。

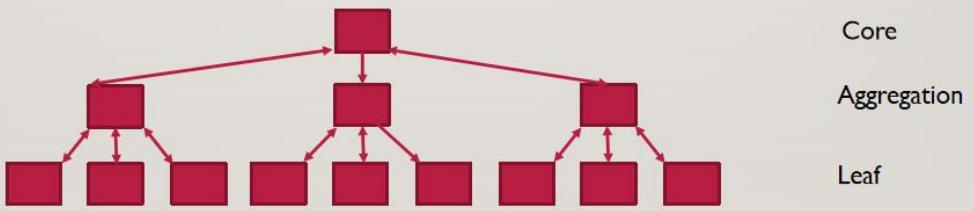
所以主要是它的逻辑够不够快。

## DC TOPOLOGY

store and forward 是交换机实现的机制之一。

### DC TOPOLOGY: VENERABLE 3-TIER

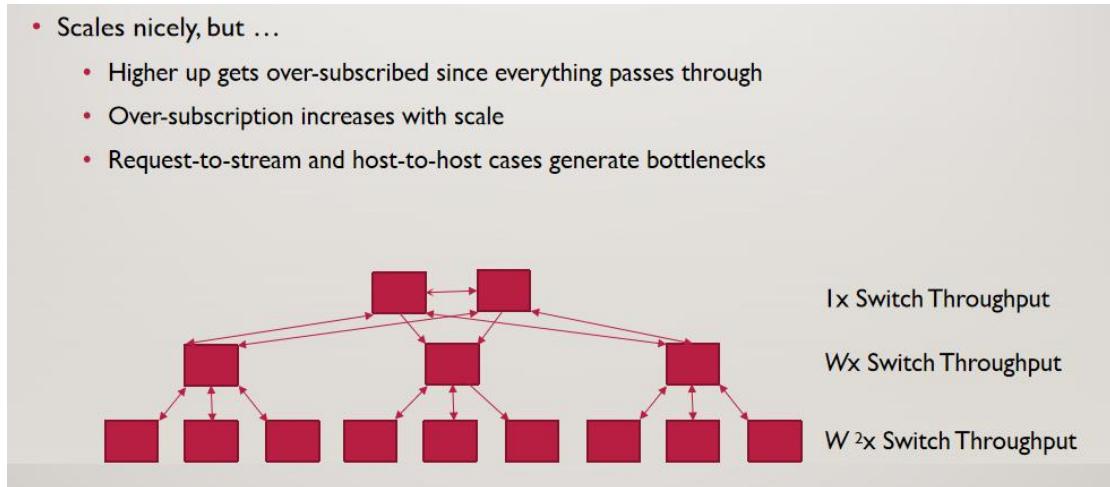
- Since, beyond a certain point, we can't make switches wider and/or faster, we need to "fan out", most commonly with a tree topology
- Venerable 3-tier network is a straight-forward example:



这就是我们 DC 的树形拓扑。在一般的 DC 中，一般这样三层就够了。还有一个名词是 TOP OF RACK，机架顶部有一个交换机，机架中全部往上连。机架之间再通过两层连在一起。为了保证容错性、高带宽，Core 的要求多一些。理论上 Core 的压力是最大的，我们的部署的时候尽可能把交互最频繁的节点放到叶节点。所以通常 Core 会部署多个来分流。TCP 有对应的 MPTCP(Multi-path TCP)，可以把多路径的资源捆绑在一块使用。为了一个应用捆绑多个带宽是之前的 TCP 做不到的。

## 三层的交换机制 fan-out (可扩展性)

- Scales nicely, but ...
  - Higher up gets over-subscribed since everything passes through
  - Over-subscription increases with scale
  - Request-to-stream and host-to-host cases generate bottlenecks

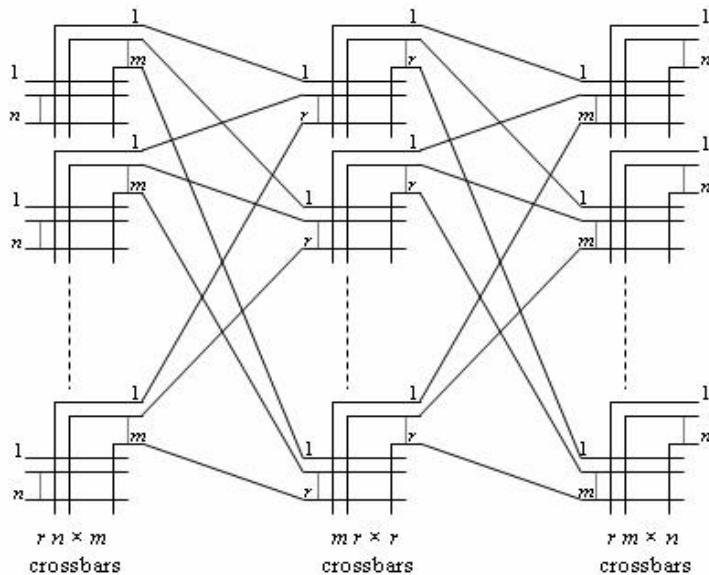


一般情况下会有 Over-subscription，也就是订阅更多的资源和部署更多的资源。但是过多和过少的都不好，最好的情况下都是部署对应资源然后 100%。需求一定会根据 scale 增加，我们一般会部署一定的余量。

传统 DC Core 的网络带宽利用率只有 5%，一旦达到 20% 就有用户会投诉。原因就是要保证服务质量，类似打电话拨号一次接通的概率为 99.7%。

## CLOS Network

这个和 CrossBar 类似。网络的思想是从 CLOS Network 中来的。我们需要一个  $N \times N$  的 network，肯定是要从小的搭建出来。用远远小于  $N$  的  $m$ （例子中为 3）。



用小的模块（一进三出）来组成。每一个小块的功能是一样的，只是连接方式不一样。这样层层叠加就用很小的  $m$  个交换，搭出一个  $N \times N$  的模块。而且它最大的问题是跳数一样，

过来通过的线数是一样的。所以我们交换机四个口非常便宜，16~32 口的就很贵了。

这就是 DC 中的用便宜的小交换机达成大的节点。这就搭成了我们 DC 的 leaf and spine 结构。

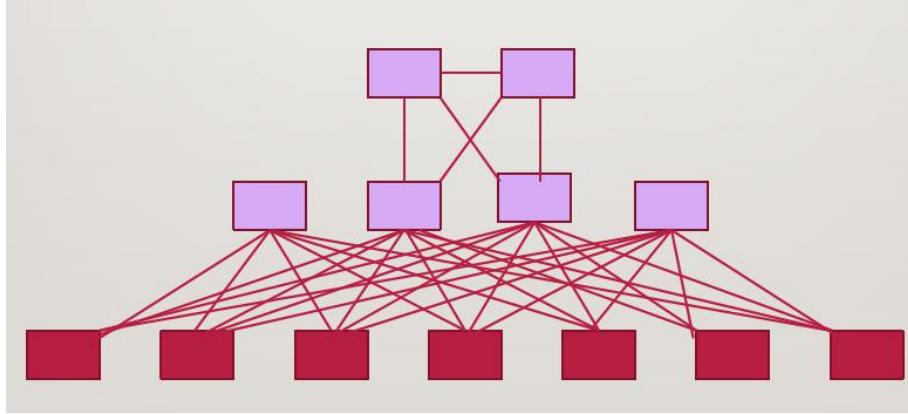


图 4 Leaf and Spine

下面就是连接了我们的电脑，我们可以看到下面有很多冗余的路径，但是跳数是一样的，所以我们就有了 MPTCP 协议。这种机制利用了 CLOS 的网络拓扑结构。

特点：

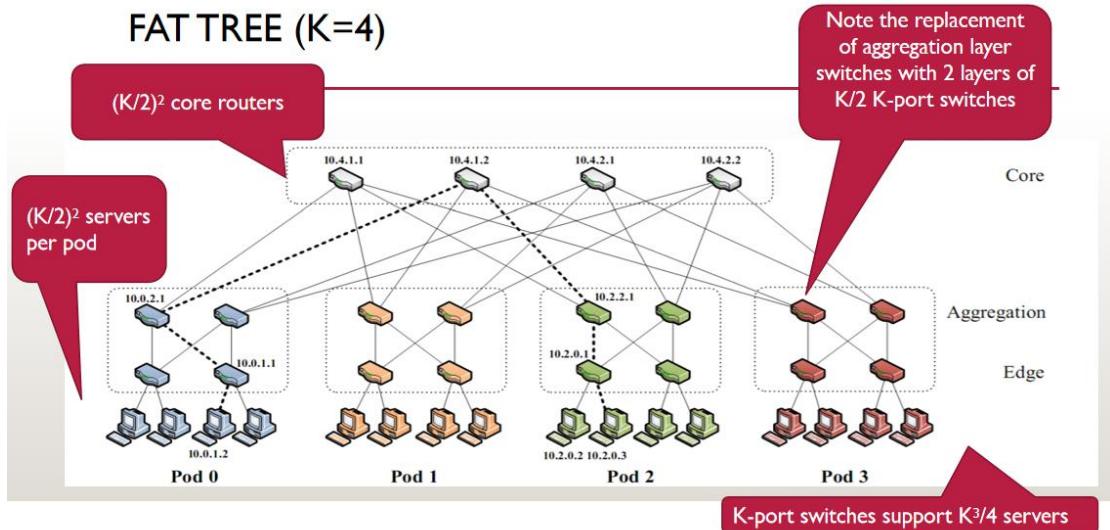
1. 所有的边缘到另一个边缘，路径是一样长度。
2. 对于 Switch vendor 只需要提供一个便宜的小交换机搭建即可。
3. 在冗余路径上需要做 balance。
4. 它可以在 2 层也可以在 3 层上。

所以我们刚才看到的东西还有一个名字就是 Fat Tree，越往上越高层，数据流的汇聚效果越明显。这里面就有另一个问题，带宽线性增加，价格指数增加，所以我们不能依赖于高性能交换机，只能用便宜的小交换机。

## 便宜路由器的胖树的目标

### FAT-TREES WITH SKINNY SWITCHES: GOALS

- Use all commodity switches
- Full throughput from host-to-host
- Compatible with usual TCP/IP stack
- Better energy efficiency per unit throughput from more smaller switches than fewer bigger switches



假设一个小交换机有 4 个口，Aggregation 有两层，每层是  $\frac{K}{2}$  个，一半朝上连接一半朝下连接。往下可以连接  $\frac{K}{2}$  (单个朝下的线)  $\frac{K}{2}$  (个)  $K$  (个 pod)，向上的线也是这么多。上面每一个人都可以提供  $K$  个 Port，所以我们需要  $\frac{K}{2}$  (单个朝下的线)  $\frac{K}{2}$  (个)  $K$  (个 pod)  $\frac{K^2}{4}$  个核心路由器。

## FAT TREE DETAILS

- K-ary fat free: three layers (core, aggregation, edge)
- Each pod consists of  $(K/2)^2$  servers and 2 layers of  $K/2$  K-port switches.
- Each edge switch connects  $(K/2)$  servers to  $(K/2)$  aggregator switches
- Each aggregator switch connects  $(K/2)$  edge and  $(K/2)$  core switches
- $(K/2)^2$  core switches, each ultimately connecting to  $K$  pods
  - Providing  $K$  different roots, not 1. Trick is to pick different ones
- K-port switches support  $K^3/4$  servers/host:
  - $(K/2 \text{ hosts}/\text{switch} * K/2 \text{ switches per pod} * K \text{ pods})$

如果一个包属于这个 pod 就往下走，如果不属于这个 pod 就往上走。

## Portland Address

现在我们提出另一个问题：数据中心网络中，一个机器坏了，我们知道它的 IP 地址和 Mac 地址了，然后怎么做呢？在 DC 中，我们需要知道机器的位置信息，但是我们讲 IP 的时候，从来没有用 IP 代表过地址。

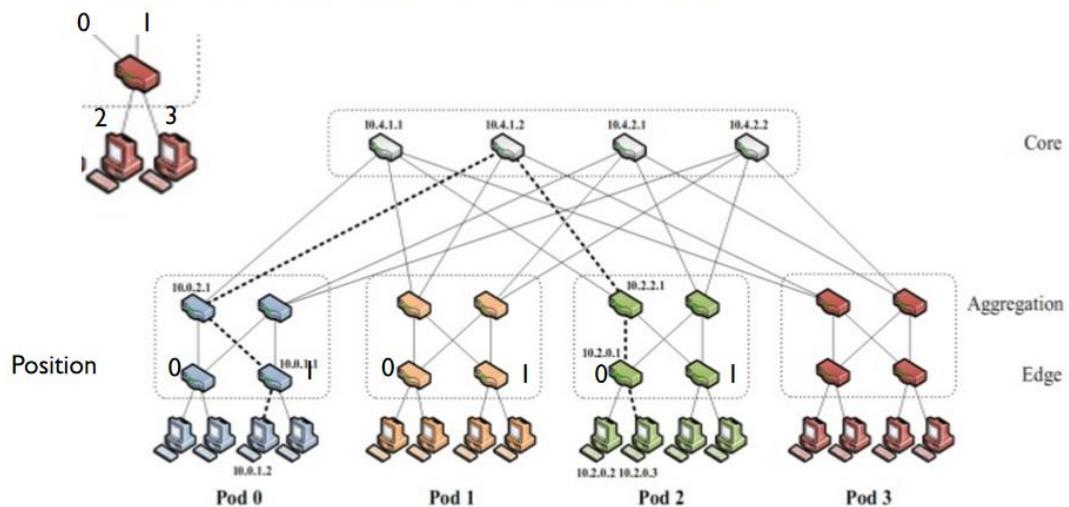
所以我们要把 IP 和位置有效地表现出来才能有效地管理。传统的 Mac 地址是出厂的时候芯片自带的编码，我们现在要标志出来就要加一层使其代表一定的物理信息。这样我们就知道是第几排第几列的服务器。

解决方案就是波兰地址，其实就是 pod:position:port (找到了服务器) :vmid (服务器上的虚拟机是哪个)。

## PORLAND ADDRESSES

- Normally MAC addresses are arbitrary – no clue about location
  - IP normally is hierarchical, but here we are using it only as a host identifier
  - If MAC addresses are not tied to location, switch tables grow linearly with growth of network, i.e.  $O(n)$
- PortLand uses hierarchical MAC addresses, called “Pseudo MAC” or PMAC addresses to provide for switch location
  - <pod: position: port: vmid>
  - <16, 8, 8, 16> bits

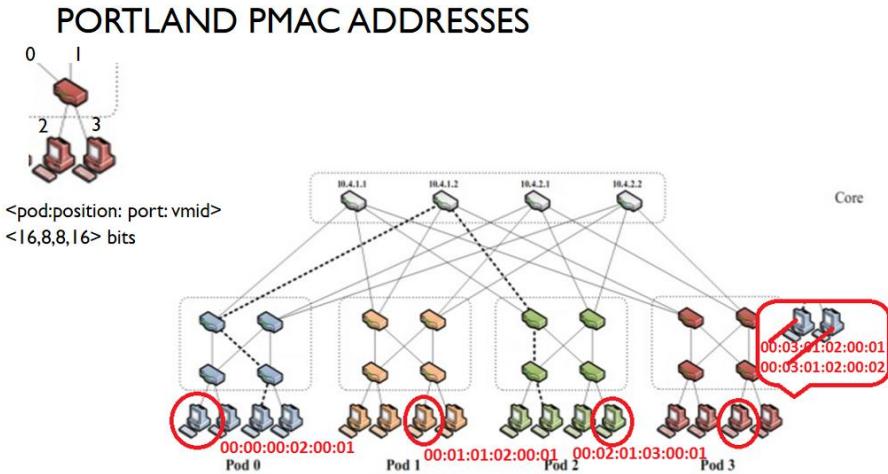
## PORLAND PMAC ADDRESSES



PMAC: <pod.position.port.vmid>

48 bits: <16-bits.8-bits.8-bits.16-bits>

分为纯软、SRIO、Para-virtualization。Data-path 就是把数据给过来并且有中断，实现 event 机制。



每个 Host 不换，还是使用物理 Mac 地址，而 Switch 把这个换一下。所以网络位置信息的保存就需要通过交换机来做。进入我们的虚拟机以后，地址就变成了实际的 Mac 地址。所以虚拟机对于这些位置信息是无感透明的。

边缘的交换机（叫做 Fabric Manager）就要做这张表。

## NAME RESOLUTION: MAC→PMAC→IP

- End hosts continue to use *Actual MAC (AMAC)* addresses
  - Switches convert PMAC  $\leftrightarrow$  AMAC for the host
- Edge switch responsible for creating PMAC:AMAC mapping and telling Fabric Manager
  - Software on commodity server, can be replicated, etc. Simplicity is a virtue.
  - Mappings timed out of Fabric Manager's cache, if not used.
- ARPs are for PMACs
  - First ask fabric manager which keeps cache. Then, if needed, broadcast.

假如我们的机器从一个 pod 迁移到了另一个 pod，我们的 AMAC (actual mac) 地址和 IP 都没有变，只是我们需要把 PMAC 通过新的对应的上层交换机重新改一下就行了。

## VM MIGRATION

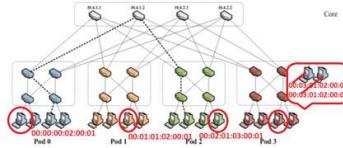
- Flat address space.
- IP address unchanged after migration, higher level doesn't see state change
- After migration IP $\leftrightarrow$ PMAC changes, as PMAC is location dependent
- VM sends gratuitous ARP with new mapping.
- Fabric Manager receives ARP and sends invalidation to old switch
- Old switch sets flow table to software, causing ARP to be sent to any stray packets
  - Forwarding the packet is optional, as retransmit (if reliable) will fix delivery

我们认为走失的数据包会对性能造成很大的抖动。怎么做走失包的处理，有一个

redirection 操作。

## LOCATION DISCOVERY: CONFIGURING SWITCH IDS

- Humans = Not right Answer
- Discovery = Right Answer
- Send messages to neighbors – Get Tree Level
  - Hosts don't reply, so edge only hears back from above
  - Aggregate hears back from both levels
  - Core hears back only from aggregate
- Contact **Fabric Manager** with tree level to get ID
  - Fabric Manager is service running on commodity host
  - Assigns ID
  - Soft state



到达某个 pod 走失数据包就重新导向到对应的目的地，这样对用户来说是无感的，可能延迟大一点。

## FAILURE

- Keep-alives like the link discovery messages
- Miss a keep alive? Tattle to the Fabric Manager
- Fabric manager tells effected switches, which adjust own tables.
- $O(N)$  vs  $O(N^2)$  for traditional routing algorithms (Fabric Manager tells every switch vs every switch tells every switch)

Fabric Manager 害怕软件路由器被病毒感染了，进行截流信息，所以 Fabric Manager 中还会要求每个 server 发送自己发送了/接收的网络包的数量。如果发收不匹配，那么可能存在截流的可能。

## LOOKING BACK

- Connectivity – Hosts can talk! No possibility of loops
- Efficiency – Much less memory needed in switches,  $O(N)$  fault handling
- Self configuring – Discovery protocol + ARP
- Robust – Failure handling coordinated by FM
- VMs and Migration – Each has own IP address, each has own MAC address
- Commodity hardware – Nothing magic.

locality 就非常简便，简单的好处就是快。self-configuration 就是说有一个 Fabric Manager 要做集中管理来掌握全局信息。

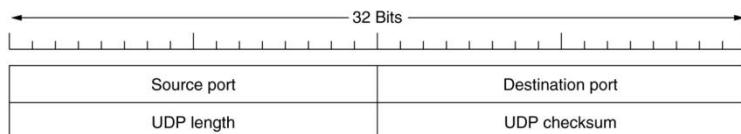
Migration 解决云平台的管理和 Failure 的问题，比如我们 4 个交换机坏了 3 个，那么我们需要挪走+修完挪回来，这就是两次 migration。

下堂课我们讲 port，也就是从局域网里抠出来一部分形成一个虚拟的局域网。

2022/3/8

TCP 对应的就是 UDP。UDP 就是一个裸露的 IP。

## UDP Header

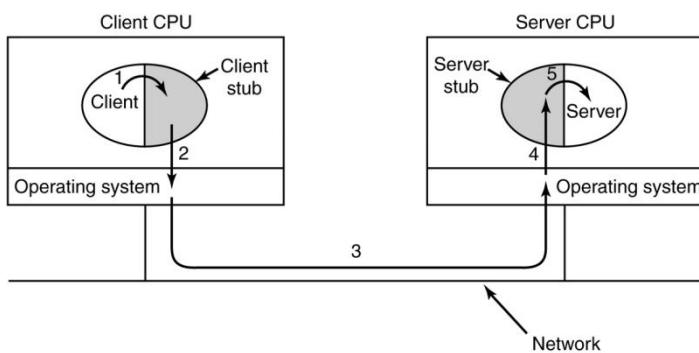


UDP 在上面没有加很多 control 的东西。

## Characteristics of UDP

- No connection establishment
- Do not perform
  - Flow control
  - Error control
  - Retransmission
- Suitable for small request/response scenario
  - E.g., DNS

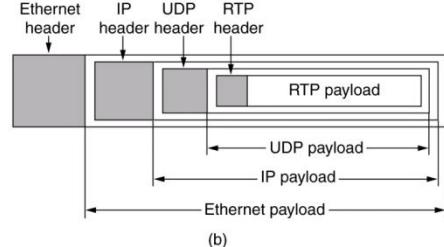
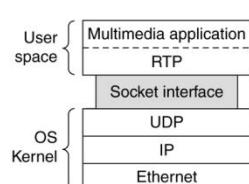
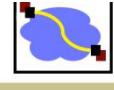
## Remote Procedure Call (RPC)



RPC 就是实现一个 RPC 接口。Server 通过注册的 RPC 的 process id 来调用远程的功能。这是分布式系统的一个很重要的元素。但是从研究的角度都不太好，因为 network、context

switch 的 overhead 都很高。

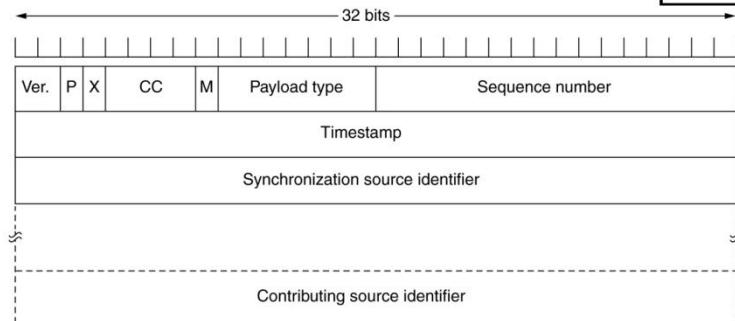
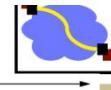
## The Real-Time Transport Protocol



- (a) The position of RTP in the protocol stack.
- (b) Packet nesting.

UDP 的包比较轻量级，适合对实时性比较高的，如网络电话。所有应用层的包头、协议，只是让大家看一下，不需要记忆。

## RTP Header



- P: padding to a multiple of 4
- X: extension header
- CC: contributing sources
- Payload type: encoding algorithm
- Sync source id: stream ID

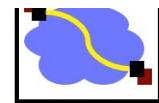
## TCP

### Transmission Control Protocol (TCP)

- TCP Segment Header
- TCP Connection Establishment & Release
- TCP Connection Management Modelling
- TCP Transmission Policy
- TCP Congestion Control
- TCP Timer Management
- Transactional TCP

TCP 就要实现 UDP 不管的 control。再往上就是有带事务的 TCP。

## TCP Basics



- 16-bits port
  - Well-known ports < 1024
- Connections:
  - Full duplex and point-to-point
  - Byte stream, not messages
- Urgent data
  - E.g., when user hit DEL or CTRL-C, send all data
  - Receiver is interrupted (signaled)

# Some TCP Services



Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial File Transfer Protocol
79	Finger	Lookup info about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

2022/3/11

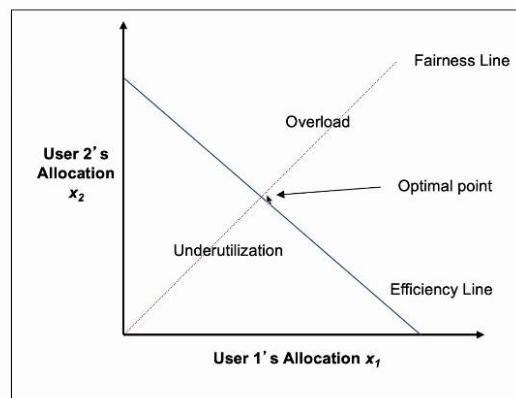
TCP 从端到端的 Control 上更多注重的是多人端到端的情况下对网络的控制。

1. 要解决全部带宽高利用率的问题
2. 各个流之间要有一个公平性的问题

## Phase Plots

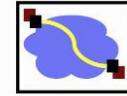


- What are desirable properties?
- What if flows are not equal?

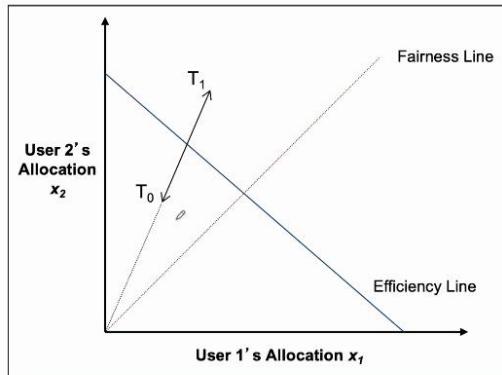


也就是流要达到之间的这个交叉点 optimal point。

## Multiplicative Increase/Decrease



- Both  $X_1$  and  $X_2$  increase by the same factor over time
  - Extension from origin – constant fairness

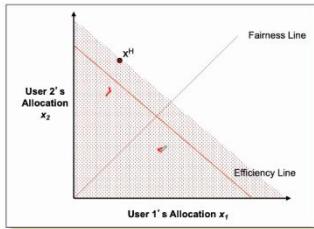


有了这两种趋势，我们才有 AIMD 这种说法。

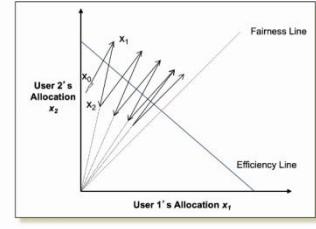
### Convergence to Efficiency



### What is the Right Choice?



- Constraints limit us to AIMD
    - Can have multiplicative term in increase (MAIMD)
    - AIMD moves towards optimal point
- Additive Increase Multiplicative Decrease



这两个点在任意位置处。只要有线性增加和指数减少的趋势，就会慢慢在这个区间上抖动，使得至少在 optimal point 附近进行变化。这就是 TCP 端到端做网络流量控制和冲突避免的主要的机制来源。

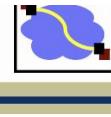
## Important Lessons



- Transport service
  - UDP → mostly just IP service
  - TCP → congestion controlled, reliable, byte stream
- Types of ARQ protocols
  - Stop-and-wait → slow, simple
  - Go-back-n → can keep link utilized (except w/ losses)
  - Selective repeat → efficient loss recovery
- Sliding window flow control
- TCP flow control
  - Sliding window → mapping to packet headers
  - 32bit sequence numbers (bytes)

最后这些都讲过了。sliding window 主要讲了 flow control。

## Other TCP Timers

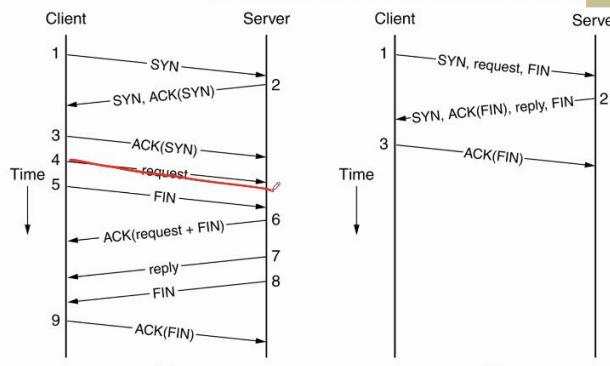


- Persistence timer: to probe receiver's window size
- Keepalive timer: idle for a while, check if peer is alive
- *TIMED\_WAIT* state timer: 2 max packet lifetime

timer 基本上是硬件实现的。还有一种 timer 就是 feed dog。在 timer 倒计时归零之前，要设置成另外一个值。这样就可以保证我们的程序是活着的，如果我们的程序被恶意程序劫持了，那么劫持了以后只要到时间不去喂狗，这样系统就会重启，或者做一些更复杂的行为。重启一般情况下会把恶意代码的 routine 的权限收回。

然后上堂课最后讲了 request

## Transactional TCP (T/TCP)



- (a) RPC using normal TCP.
- (b) RPC using T/TCP.

三次握手，传递信息以后关闭 TCP。short-life session 在网络中是很重要的问题。所以就有人把 request-reply 在建立连接的时候同时发出来，但是这种情况对于 ddos 攻击是非常好的情况，它只需要在第一步发消息就可以了。那么我们整个系统负担还是很重的，因为我们要建立连接、reply 再关闭连接。所以 server1 可以以很小的代价对 server2 产生很大的开销。ddos 可以劫持很多网络中的嵌入式芯片发送请求，我们的服务器很快就会崩溃。

上一堂课的 DCA 部分还剩下的一点东西。它本身就是在 TCP 上建设的叠加网络。

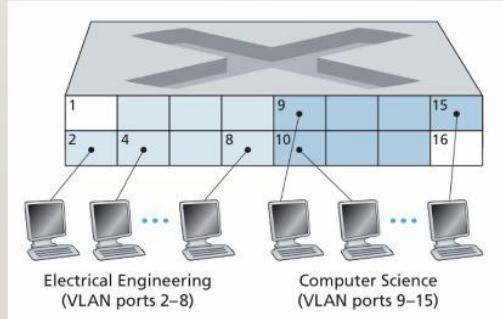
# VLANS: BIG PICTURE BENEFIT

- Build out one large switched network
- Configure it to act like any number of LANs
  - But without any of the geographic limitations

我们先来看 VLAN 的好处是什么，为什么我们有了局域网之后还需要一个虚拟的 LAN。它的目标很明确。

## VLANS: PORT-BASED

- *Static VLAN: VLAN=Group of Ports*
  - Port = switches' wire connection
- Two VLANs configured on a 16-port switch
- How do the VLANs communicate with each other?

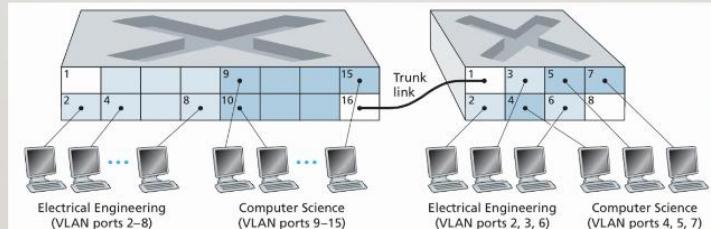


比如在网络里发一个广播（对所有网段里的网口发一个消息），它的物理层协议是支持的。但是我们不想让硬件决定接收对象，比如有 16 个网口，我们希望对 2~8 和 9~15 分别发送消息。我们假设不分开这两个 VLAN 的话，对于逻辑上 2~8 的广播，9~15 就是无用的信息。

我们可不可以加一个 VLAN 使得发给 9~15 的信息只被 9~15 的网口接收。有了 VLAN 之后，并且交换机有协议支撑，就不会把发给 9~15 的信息给 2~8 听到。

## VLANS:TRUNKED SWITCHES

- Trunked connection: port belongs to all VLANs → all frames at that port are forwarded to all VLANs

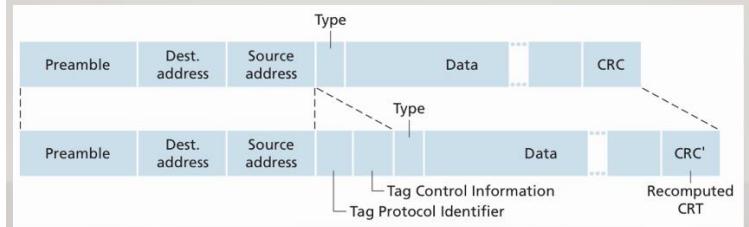


- But, how does the receiving side know which VLAN a particular frame belongs to?

除了分两组硬的，还可以有 Trunked connection。这个图里，有一个点就是要听到所有点的消息，就是把两个 Switch 连成了 24 口的交换机。按照颜色的深浅来说，Switch1 的 2~8 和 Switch2 的 2,3,6 可以分成一个 group，中间的桥接通过 16 来进行连接。

## 802.1Q TAGGED ETHERNET

- VLAN identifier added to Ethernet frame
  - 4-byte VLAN tag
  - Includes 12-bit VLAN identifier
- Sending switch adds tag, receiving switch parses and removes tag



Net 里的地址转换，其实在 TCP/UDP 的包头里面来做。但是我们看 802.1Q 是在以太网里打了一些 tag。tag 在 SDN 领域，我们就知道它大致在做什么事情。VLAN 是非常经典和简单的一个例子，有一些 SDN 的交换机里可以非常灵活地加各种各样的 tag。很多协议栈有一些空位，可以在这些位置上打一些标志，使得数据包在网卡接收端、交换机上直接做一些事情。

比如我们为了让 CPU 少做软件的 bridge（开销很高），如果我们在 tag 里打上一些 id，我们就可以更好地推送到一些虚拟机和容器中。这样中间的转换流程就会很方便，甚至我们可以使用 tag 做一些流控制。

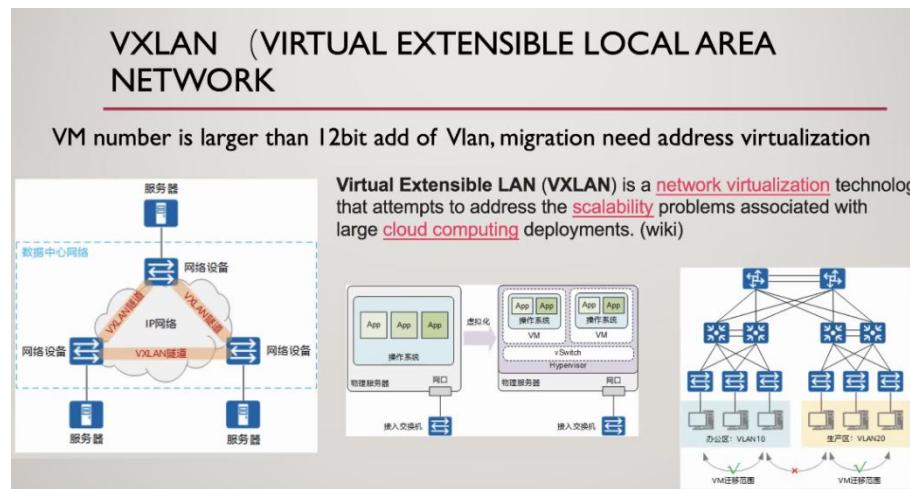
它就是在这个位置上塞了 Tag Protocol ID 和 Tag Control Information，做了一些标志。接收端就会把 tag 解析出来做一些行为，并且删掉，使得 destination 是一个标准的以太网消息。

这里面我们出现了十二位的 id（4000 多），一开始 4000 多对于 VLAN 是足够用的。但

是最终到了数据中心里，对于一些特别大型的数据中心中，其中一台物理机上分成了几十个虚拟机，此时 4K 就不太够用了。

802.1Q		VLAN数据帧 6Byte Destination address 6Byte Source address 4Byte VLAN Tag 2Byte Length/Type 46-1500Byte Data 4Byte FCS
字段	长度	含义
Destination address	6字节	目的MAC地址。
Source address	6字节	源MAC地址。
Type	2字节	长度为2字节，表示帧类型。取值为0x8100时表示802.1Q Tag帧。如果不支持802.1Q的设备收到这样的帧，会将其丢弃。
PRI	3比特	Priority，长度为3比特，表示帧的优先级，取值范围为0~7，值越大优先级越高。用于当阻塞时，优先发送优先级高的数据包。如果设置用户优先级，但是没有VLANID，则VLANID必须设置为0x000。
CFI	1比特	CFI (Canonical Format Indicator)，长度为1比特，表示MAC地址是否是经典格式。CFI为0说明是标准格式，CFI为1表示为非标准格式。用于区分以太网帧、FDDI ( Fiber Distributed Digital Interface ) 帧和令牌环网帧。在以太网中，CFI的值为0。
VID	12比特	LAN ID，长度为12比特，表示该帧所属的VLAN。在VRP中，可配置的VLAN ID取值范围为1~4094。0和4095协议中规定为保留的VLAN ID。三种类型： <ul style="list-style-type: none"> <li>■ Untagged帧：VID 不计</li> <li>■ Priority-tagged帧：VID 为 0x000</li> <li>■ VLAN-tagged帧：VID 范围0 ~ 4095</li> </ul> 三个特殊的VID： <ul style="list-style-type: none"> <li>■ 0x000：设置优先级但无VID</li> <li>■ 0x001：缺省VID</li> <li>■ 0xFFFF：预留VID</li> </ul>
Length/Type	2字节	指后续数据的字节长度，但不包括CRC检验码。
Data	42~1500字节	负载（可能包含填充位）。
CRC	4字节	用于帧内后续字节差错的循环冗余检验（也称为FCS或帧检验序列）。

这里面我们也不用记忆这 12bit 的 VID 在做什么。最主要的就是这个 VID 不够用，所以就有了这个 X。



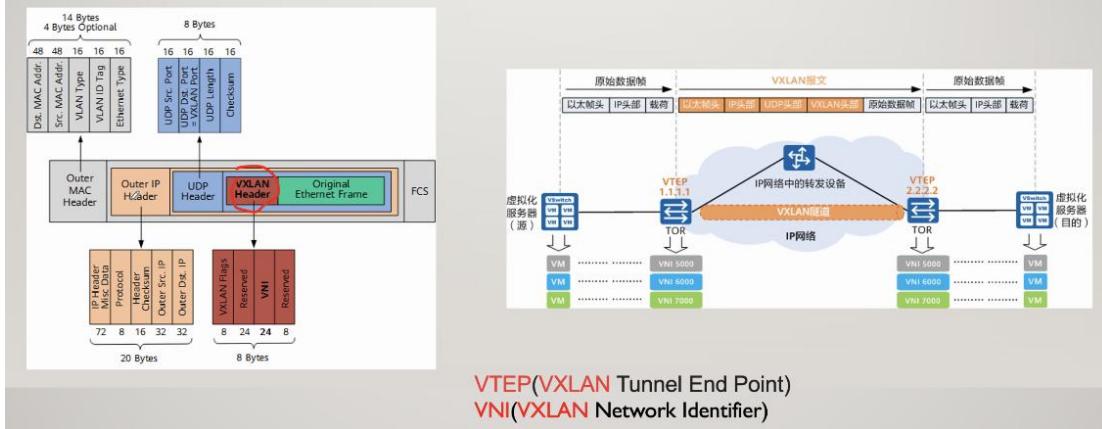
原因就是 12Bit 不够用，虚拟化的情况下 vSwitch 负责分发给不同的虚拟机。这样我们就可以在一台服务器上配置很多虚拟机。

数据中心网络有了这些 VLAN 以后，就可以建立一些隧道。阿里、华为有成千上万的服务器，我们是一个大客户要租几十个服务器、虚拟机。这些服务器，我们希望是分开真正属于我们自己的，分开的方式其实就是 VLAN。我们希望在我们的服务器之间建立更高效的通道、并且我们的服务器交互地更频繁，并且我们不希望听到别人服务器的数据包。这就是网络虚拟化。

每个虚拟机都会有一个虚拟网卡，这也是网络虚拟化的一部分。还有一部分就是我们要在网络上建立虚拟的通道，和别人不干扰。之前我们学了 pmarker，我们的地址会变化，在迁移之前交换机有一个 pmark。迁移之后，我们的 mark 是不变的，只是在边缘交换机上的映射变化了，这样对上下都是无关的。

这样 VXLAN Address 的格式如下：

## VXLAN ADDRESS FORMAT, VTEP, VNI



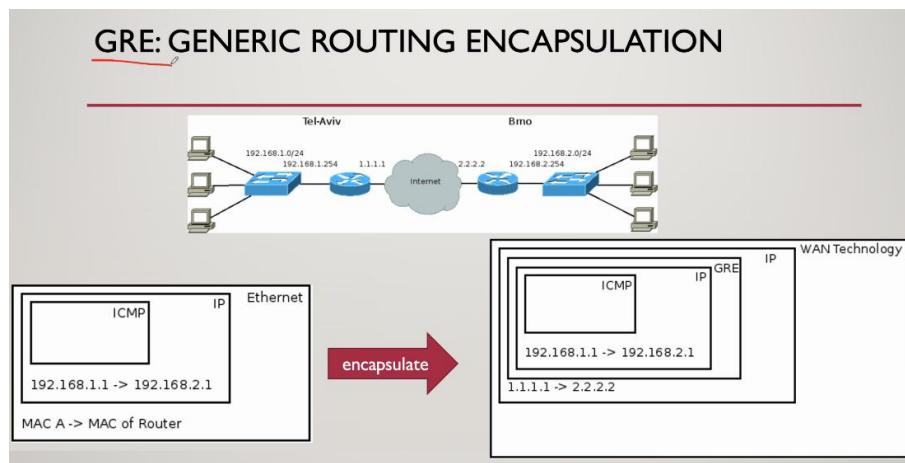
它在 UDP 里面，所以它叫 **overlay**。我们在 IP 网络里面又架了一层虚拟的 IP 网络。而且这一层完成的也是 IP 层的任务，但是它是完全虚拟的，这里面有一些 **header** 的缩写。这里面有一些 **Reserved** 的，可以做一些额外的工作。只要有 SDN 交换机上有识别的程序，比如 **match action**（包符合什么条件以后，做什么事情）。**matching** 是完全硬件化的，比如在包头有一块 **tag** 以后做什么事情、调用什么函数。我们就可以在保留位上注册一些 **match action**。函数无非是对包做一些处理。

**hyperscan** 可以对流做分析。对 UDP, TCP 的数据流建立连接的一些行为特性建模，比如用户连接的游戏网站、搜索网站的行为特性建模。这就需要一些 AI 的方式，对于这样的数据流，我们可以给予不同的优先级、流量控制、**denial service** 等。

**VTEP** 其实就是一个隧道技术。**Net** 也是一种隧道的技术。**VLAN** 也是一种隧道的技术。这里面有大量的虚拟机，每个虚拟机对应的是一个 **VNI**，**VNI** 通过隧道（跨云、跨机架的方式）。

这个图的特性和外网访问到内网很像，虚拟机的 **mark** 地址和 **VNI** 也很像。所以 **VLAN** 完全解决了 **Net** 的功能并且是 **Net** 的变形。

很多人现在已经模糊化了隧道技术和 **Net** 技术，因为都是一种 **mapping**。



然后除了 **VLAN** 以外，还有一种 **GRE**。有人认为 **GRE** 更适合做路由阶段，而 **VLAN** 更适合虚拟机的 **mapping**。**GRE** 就是以太网包头，**IP** 包头加了 **GRE**，再加了一层 **IP**，简单来说就

是 IP 套 IP。VLAN 直接使用 VID 来管理。

同样，GRE 也可以实现内外网的地址转换。地址用的都是局域网的地址。1.1.1.1 和 2.2.2.2 是大型的局域网的 IP，可以认为是广域网 IP。我们在此之中要进行多级的 IP 地址转换。IP 不再是 socket 那套东西了。IP 要想到达 destination，必须拥有隧道技术才能过去，不进行地址转换根本没有办法到达最终的地址。

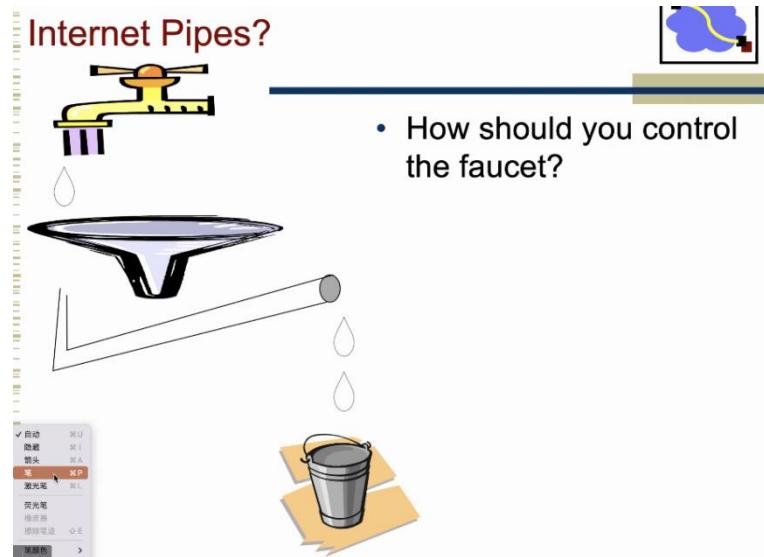
这就是为什么把 DCN 的这些往后放一放，因为这些是放在 UDP 层里头的。很多互联网公司做云计算，它们某些公司 VLAN 和 GRE 其实是可以通用的，但是它们一般只会选择一种。都是在 IP 层之上，VLAN 套在 UDP 里，GRE 套在 IP 里面。

## QoS

我们前面讲的 TCP，它的 QoS 主要是端到端的。VLAN 虚拟的建立了一个隧道。这堂课讲的和前面就不一样了，在中间我们也得做一些事情才能更好地服务器。有一本教材先讲两端为主，因为很多做软件的都是在两端做。但是现在云商肯定要做一些中间的交换机的配置和 QoS 的配置。所以这节课关注的是中间怎么做的。TCP 扩展到多个用户是怎么做的，为什么大家都是慢速增长的情况下，大家再除以 2，这样就可以到达最终的平衡点。

为什么 TCP 最小的，比 link 层还会更好。网络里既有用户刚刚减半的情况，刚刚进入网络指数增长的情况下，可能比起均匀的情况要更好。

为什么拥塞会产生非常恶劣的影响，所以 router 才会做 control。从 QoS 来说，从中间能做什么事情？



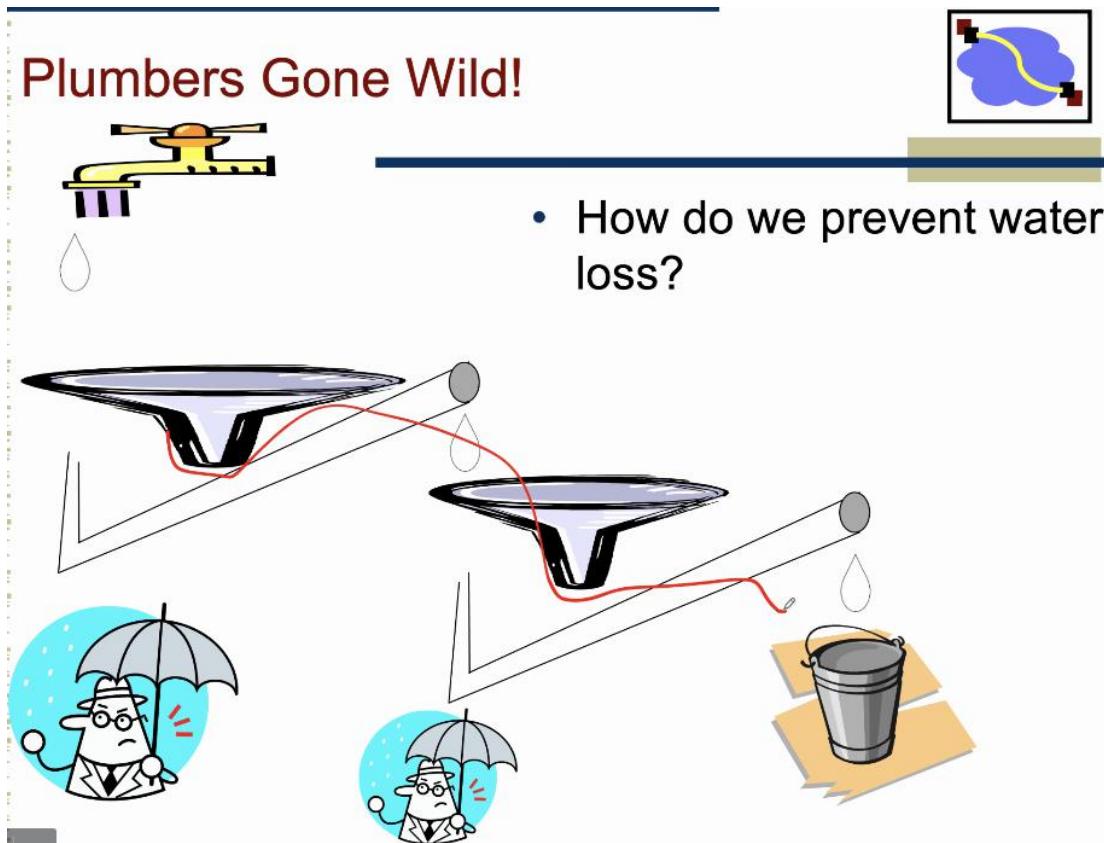
举个例子来说，水龙头和一个洗手池假设拥塞了，池子里的水就会越来越高。一旦水位高了，水池里的水就会溢出。从水龙头到最终的桶上能够传递多少。溢出其实就是丢包，丢包不一定是物理层 01 翻转并且不能纠偏，更多的情况下是中间的路由器处理不完，队列满了就把包扔掉了。它默认就算传到了也会超时。

我们要避免几种情况：

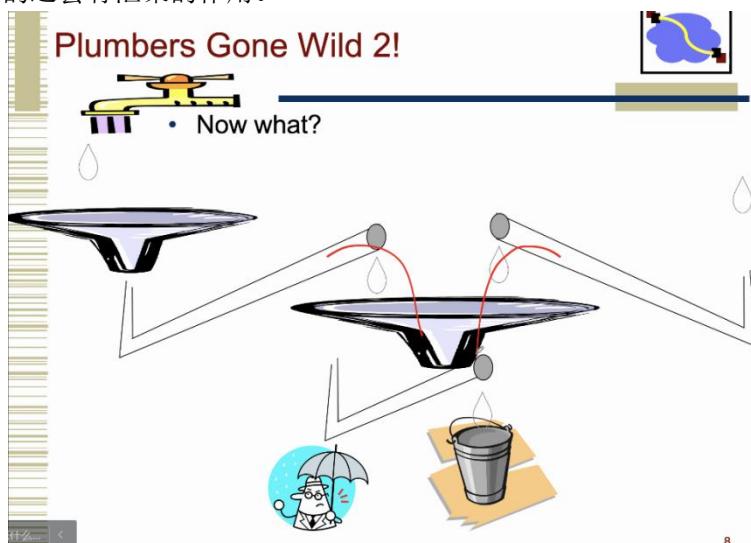
1. 不能太多，否则就会慢
2. 不能太慢，太慢的带宽利用率就太低了。

我们的目的是让它尽可能快地接到一桶水，并且水在水池里还不溢出来。我们现在更多

讲的是中间的水池应该怎么调节的问题。我们不再讲 TCP 端到端的 flow control 机制。

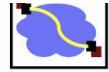


每一个交换机有大有小。不同交换机 MTU 是不同的。MTU 大一些的，传输带宽也会大一些。更复杂的还会有汇聚的作用。



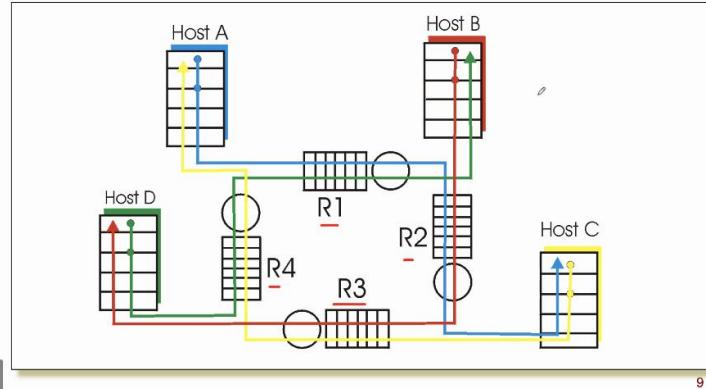
这和 TCP 的发送端接收端 1 对 1 建立的 session 就不同了，路由器端要处理汇聚的情况。因为路由器不能只为你一个人服务，所以我们不能控制其他控制包什么时候过来。但是我们可以看到食堂里排队、坐着的有多少人。这里给了很简单的例子。

## Causes & Costs of Congestion



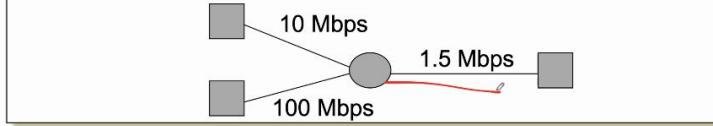
- Four senders – multihop paths
- Timeout/retransmit

Q: What happens as rate increases?



有 4 个发送端（host）和 4 个路由器。任何一个发送端都只和两种颜色碰到过。每一跳上的 traffic 都是不同的。

## Congestion

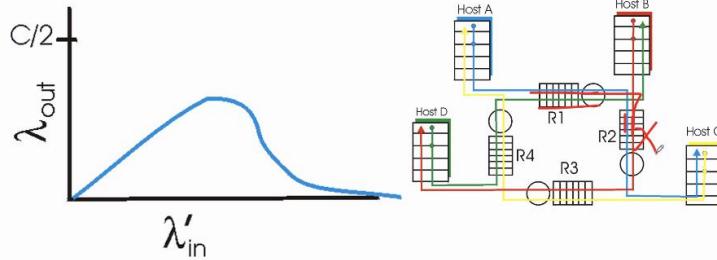


- Different sources compete for resources inside network
- Why is it a problem?
  - Sources are unaware of current state of resource
  - Sources are unaware of each other
- Manifestations:
  - Lost packets (buffer overflow at routers)
  - Long delays (queuing in router buffers)
  - Can result in throughput less than bottleneck link (1.5Mbps for the above topology) → a.k.a. congestion collapse

路由器的 1.5M 太容易把 10M 和 100M 拥塞掉了。比如现在台式机上的以太网口基本上都是 1G 的，它连到一个小的路由器再插到墙上的网口上。墙上的 IP 带宽也就是几十 M。如果每个人的请求都是按照台式机上的 1G 来说，一个人就可以灌满。简单来说，拥塞非常容易发生，即便是 2 个 1M 也很容易大于 1.5M。

我们这种情况下很难知道另一个人现在在发多少个 Traffic。所以在路由器上更多的是一个被动的方式，两个人一旦开始竞争造成 overflow，就把一些包丢掉和 timeout。通过这些事件的产生去反向地体验到 congestion，这种就是 implicit 的。也就是别人不愿意理你，就是发生了 congestion。

## Causes & Costs of Congestion



- When packet dropped, any “upstream transmission capacity used for that packet was wasted”!

为什么 **congestion** 影响范围非常大，非常不好。一旦有了丢包以后，所有的 **upstream** 的带宽和使用的 CPU 资源都是浪费的。比如 100 跳传到了 99 跳然后丢包了，那么前 99 跳都是浪费的。互联网初期，认为这是比较重要的，所以希望路由器做重传，不要全部浪费掉。但是只是说，现在的互联网/数据中心网络里，这种就没有必要了，重传就完事了。所以这也是要考虑一定的物理特性的。

左边的图是什么概念呢？其实就是每个路由器上都是 2 个人竞争，并且我们假设了每个 **host** 发送和接收的数据流都是  $C/2$ ，那么我们每个数据流对输入增加的时候，刚开始会以比较好的形式进行增加。到一定程度以后就到达了最优点，就是所有人都发  $C/2$ ，并且带宽利用率 **100%**。我们到达 **50%~70%** 以后，再往上就会发现所有人再增加了以后，**out** 就不增反降了。所以冲突不是在利用率到达 **100%** 的时候发生的，而是在远远小于 **100%** 的时候发生的。这也是一些实时网络设置在带宽利用率设置在 **70%** 以下或者 **30%** 以下的原因，这是 **real-time scheduling** 的研究范畴。

## Congestion Collapse

- Definition:** *Increase in network load results in decrease of useful work done*
- Many possible causes**
  - Spurious retransmissions of packets still in flight**
    - Classical congestion collapse
    - How can this happen with packet conservation
    - Solution: better timers and TCP congestion control**
  - Undelivered packets**
    - Packets consume resources and are dropped elsewhere in network
    - Solution: congestion control for ALL traffic**

因为我们冲突是和别人造成的，并且别人我们可以不了。并且我们不能通过简单的流量相加小于物理带宽来判断拥塞。远远小于带宽的情况下还是会拥塞。一旦拥塞了以后就会出现 **collapse**。输入都是  $C/2$  的情况下，输出可能小于  $C/4$ 。这样整个系统就崩掉了。

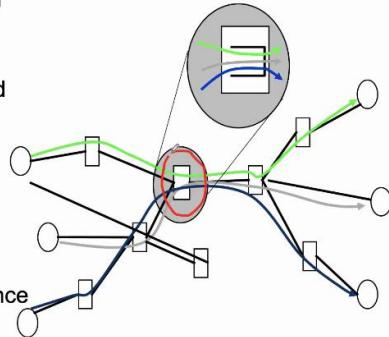
Q: 我们怎么避免它？

A: 在两端做，TCP 那边。但这堂课我们关注在中间做。

## Traffic and Resource Management



- Resources statistically shared
- Overload causes congestion
  - packet delayed or dropped
  - application performance suffer
- Local vs. network wide
- Transient vs. persistent
- Challenge
  - high resource utilization
  - high application performance



灰色的就非常容易产生拥塞的点。发送端是不可能知道这个可能产生拥塞的路由器存在的，很难在发送端做一个流量控制使得发送端和可能出现拥塞的点产生共鸣。拥塞分成集中，长期拥塞和瞬时拥塞。长期拥塞不是我们能解决的，唯一解决方法就是买更好的带宽服务。我们只能解决瞬时拥塞，是从一个点出发还是从宏观的网络层出发。从宏观出发的话，是可以做一些重路由的（work load balance）。但是上面这张图的情况比较特殊，灰色是始终要汇聚的。

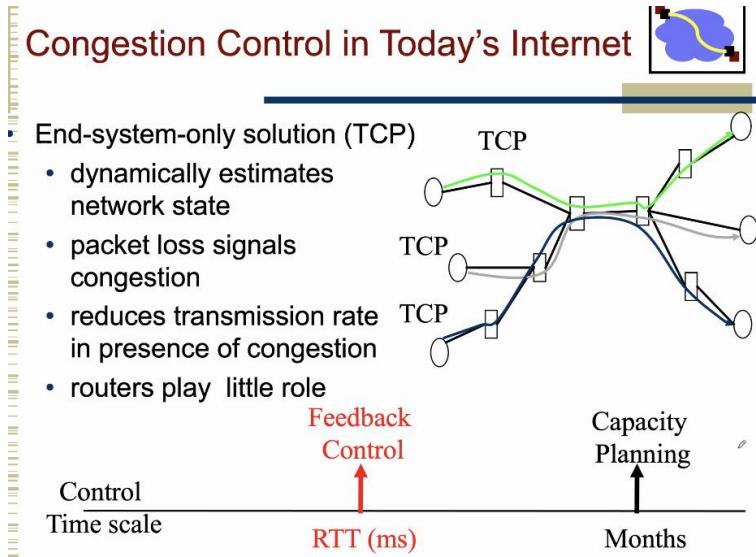
## Resource Management Approaches



- Increase resources
  - install new links, faster routers
  - capacity planning, provisioning, traffic engineering
  - happen at longer timescale
- Reduce or delay demand
  - **Reactive approach:** encourage everyone to reduce or delay demand
  - **Reservation approach:** some requests will be rejected by the network

资源管理方法：1.砸钱（增加基础设施建设）2.用优化的方式去管理资源，是研究要做的事情。主动的就是每个人 **reduce**，并且每个人要 **fairness**。保守的就是惩罚性的，在网络中对每一个人进行一定的处理（丢包），要让丢包丢得公平且温和。使得发送端和接收端感知到网络情况不好了。

TCP 其实做的就是从端到端的角度来说，如果我们丢了包，有 3 个 ack 和 timeout。前者属于丢了两个包，有 3 个 **duplicate** 的 ack 以后，就认为中间有一些地方拥塞了。



`timeout` 就是长时间没有任何的回消息，一个包都没收到。此时就认为拥塞情况比较严重。

### More Ideas on Traffic Management

- Improve TCP
  - Stay with end-point only architecture
- Enhance routers to help TCP
  - Random Early Discard
- Enhance routers to control traffic
  - Rate limiting
  - Fair Queueing
- Provide QoS by limiting congestion

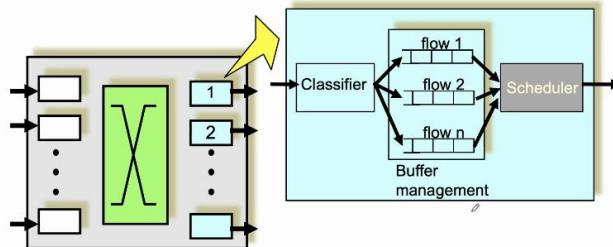
TCP 我们已经讲过了。我们看看 router 能做什么事情。比如 Random Early Discard (RED)，这是路由器必备的功能。路由器还可以做一些流量控制和公平性。只有路由器才知道有几个数据流过来并且保证它们更好地被服务。

## RED

### Router Mechanisms



- Buffer management: when and which packet to drop?
- Scheduling: which packet to transmit next?



主要是输入、输出、两大队列和中间的 switch fabric。数据包在绿色的部分中，中间还有 memory。蓝色部分数据包进来了以后，对输出的数据流按照流进行分类，因为我们有 src 和 dst，一对就认为是一个数据流。我们有客户甲、客户乙、客户丙的数据流。我们可以以 id 或者 tag 来分。然后 scheduler 有 fix priority，比如只要有 000 就服务 000，没有 000 才服务 001。这就是固定优先级的调度方法。还有的就是 round robin 的调度方法，每个 flow 服务 10ms。

### Overview

- Why QoS? -Obviously
- Queue management & RED
- QoS Principles
- Introduction to Scheduling Policies
- Integrated Services

Q: 为什么要做 QoS?

A: 毫无疑问这必须要做。

### Motivation



- Internet currently provides one single class of "best-effort" service
  - No assurances about delivery
- At internet design most applications are elastic
  - Tolerate delays and losses
  - Can adapt to congestion
- Today, many "real-time" applications are inelastic

无弹性的就是没有任何的容忍度。我们要做一个 service model。

## Why a New Service Model?



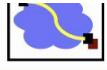
- What is the **basic objective** of network design?
  - Maximize total bandwidth? Minimize latency?

- Continuous media applications
  - Lower and upper limit on acceptable performance.
  - BW below which video and audio are not intelligible
  - Internet telephones, teleconferencing with high delay (200 - 300ms) impair human interaction
  - Sometimes called "tolerant real-time" since they can adapt to the performance of the network
- Hard real-time applications
  - Require hard limits on performance
  - E.g. control applications

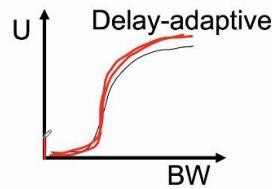
比如 hard real-time application，就是轿车的 abs。我们踩下刹车以后必须在 40ms 中完成响应。

有一些要求比较软的，就是 soft real-time application。

## Admission Control



- If  $U$  is convex  $\rightarrow$  inelastic applications
  - $U(\text{number of flows})$  is no longer monotonically increasing
  - Need admission control to maximize total utility
- **Admission control**  $\rightarrow$  deciding when adding more people would reduce overall utility
  - Basically avoids overload

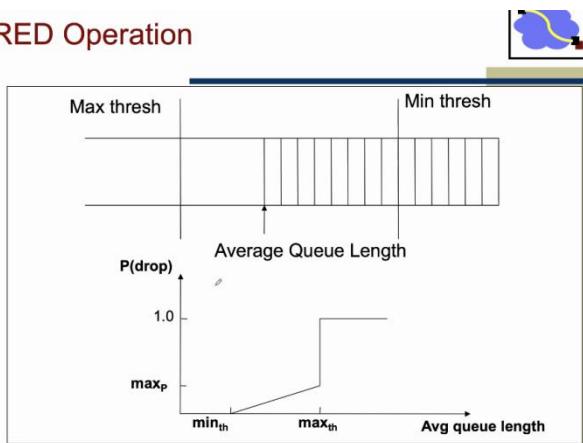


假设所有地方都是一个人，其实我们带宽利用率就接近 100% 了。假设有一个数据流对实时性要求很高，数据包必须在 100ms 送到。这时候出现了一个大象流，有一个数据包很大，一秒钟发一次，一次要发 200ms。那么此时老鼠一定会被踩死的。我们增加多少带宽能解决？增加再多也不能解决。

一旦大象开始了老鼠就会被踩死。所以在这个情况下，增加带宽就解决不了问题了。怎么侦测老鼠和大象呢？我们需要让老鼠快过去，保证大象和老鼠和平共处。简单来说，我们就谁都让进来，也不是什么时候都允许进来。

还有一个契合度的问题，最好让大象和大象在一起，老鼠和老鼠在一起。所以 RED 最重要的一点就是这张图。

## RED Operation



Random Early Detection。我们希望在队列右边的情况比较多。在中间这段有概率性地丢包，丢弃的包来自不同的数据流。

## 2022/3/15(最后 15 分钟)

我们把所有 LAB 的提交时间全部延期到疫情以后，如果有条件影响不大的话，可以往前赶进度。

上堂课我们讲到了 RED。总的来说就是 router 在中间要做些什么事情。Queue Management 对应的是算法里面的排队论章节。

## Queuing Disciplines

- Each router **must** implement some queuing discipline
- Queuing allocates both bandwidth and buffer space:
  - Bandwidth: which packet to serve (transmit) next
  - Buffer space: which packet to drop next (when required)
- Queuing also affects latency

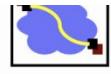
我们这里介绍的都是业界用的最多的算法。

## Typical Internet Queueing

- FIFO + drop-tail
  - Simplest choice
  - Used widely in the Internet
- FIFO (first-in-first-out)
  - Implies single class of traffic
- Drop-tail
  - Arriving packets get dropped when queue is full regardless of flow or importance
- Important distinction:
  - FIFO: scheduling discipline
  - Drop-tail: drop policy

FIFO 和 Drop tail 都统称为被动的算法，我们只能从队列的前面拿，进的时候只能从队尾进。

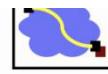
## FIFO + Drop-tail Problems



- Full queues
  - Routers are forced to have large queues to maintain high utilizations
  - TCP detects congestion from loss
    - Forces network to have long standing queues in steady-state
- Lock-out problem
  - Drop-tail routers treat bursty traffic poorly
  - Traffic gets synchronized easily → allows a few flows to monopolize the queue space

这种被动的方式其实是不好的，有一些大象流会把队列占满。运气不好的数据流一旦初始情况下没有进队列，TCP 就会降速，一旦其他人没有降速，那么就会疯狂占用队列。这样就会导致其他人被 starvation，就会有 lock-out 问题。

## Active Queue Management



- Design active router queue management to aid congestion control
- Why?
  - Router has unified view of queuing behavior
  - Routers see actual queue occupancy (distinguish queue delay and propagation delay)
  - Routers can decide on **transient congestion**, based on workload

如果传输需求就是大于带宽，这就是长期拥塞了，需要扩展基础设施来解决，所以我们讲的还是瞬时拥塞。

对网络来说，吞吐量和延迟是永远的主题，还有些人把 jitter（抖动）也考虑进去，也就是延迟差不能太大。

## Design Objectives

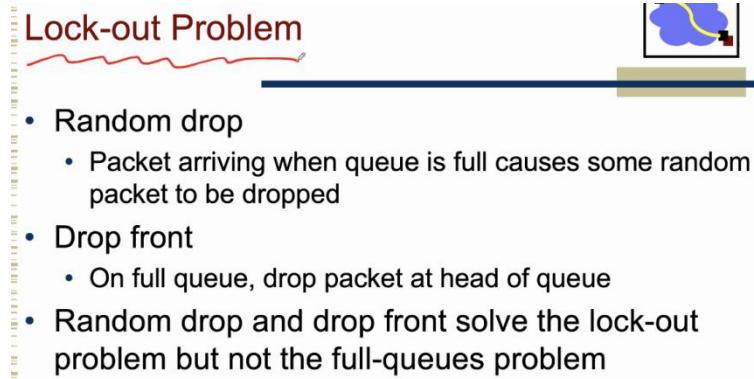


- Keep throughput high and delay low
  - High power (throughput/delay) *jitter*
- Accommodate bursts
- Queue size should reflect ability to accept bursts rather than steady-state queuing
- Improve TCP performance with minimal hardware changes

jitter 的要求相对可以使用吞吐量来弥补。Queue Size 无穷大就不会丢包，即便进了队列，

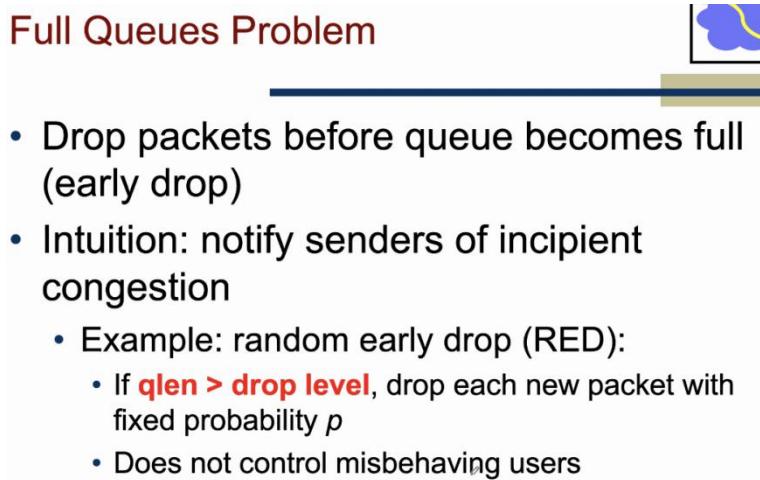
如果队列太长，再加上路由器的叠加，最终的延迟就会无穷大。所以我们希望保证队列的长度在一定的范围之内，并且对 TCP 是一个补充。

我们之前说的物理层对 Bit 的值传输错了，是误码的问题，但是我们现在遇到的更多问题是因为网络拥塞把包丢掉了。现在我们要用 active queue management 来解决死锁和 starvation。



所以目标不是队列满了以后被动地扔（会叠加在不幸的数据流上），有人认为可以 Drop Front。这种方法肯定有可以应用的地方，但是数据包排队了很长时间占据了空间到了最前面被扔掉了，所以这里面就引出了我们的 RED。

## RED(Random Early Drop)



Drop Front 也不咋地，所以就有人提出了 RED 的概念。它的目标是：观测队列的长度，设置一个概率值，随机地丢包，这样就可以保证真正地队列不会满。保证所有的人进入了队列，然后再随机丢包。但是它还是不能控制恶意的用户，比如我们就是 TCP 不降速，就会很暴力地在网络里疯狂地发数据包。所以 RED 不能解决这个问题。

## Random Early Detection (RED)

- Detect incipient congestion
- Assume hosts respond to lost packets
- Avoid window synchronization
  - Randomly mark packets
- Avoid bias against bursty traffic

Early 就是不等队列满了再做。几个 TCP 的 sliding window 恰好就是队列长度的话，就会有不幸的数据流被饿死。这里面的算法非常简单。

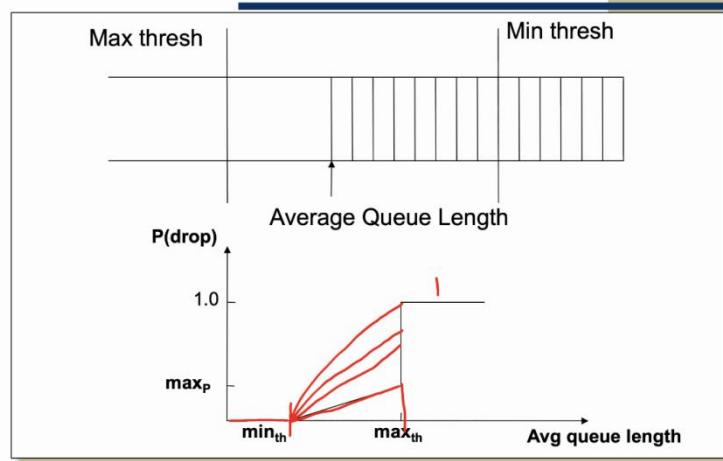
### RED Algorithm



- Maintain running average of queue length
- If avg < min<sub>th</sub>, do nothing
  - Low queuing, send packets through
- If avg > max<sub>th</sub>, drop packet
  - Protection from misbehaving sources
- Else mark packet in a manner proportional to queue length
  - Notify sources of incipient congestion

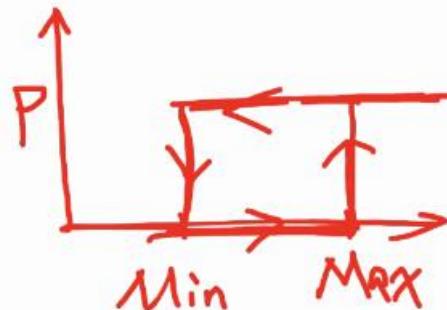
设置了两个阈值，在低阈值之前，就不丢包，在大于 max 阈值了以后，就全部丢包。在两个阈值之间，丢包概率是可调的线性值。

## RED Operation



两个概率之间的坡度是可以自己调整的。

所以 RED 讲完了以后，更多地我们是要知道它有什么好处。但是我们也做过一些研究，因为丢包是不确定的。我们就研究过，也设置两个阈值，但是丢包不是这样的。上去的时候走的是下面的这条线，开始丢包的时候走的是这条线下来。



比如我们的家里暖气就设置了 26 度，它一般加热到 27~28 度停止，温度下降到 24~25 度的时候再打开。否则电暖气开关对控制非常敏感的话很容易就坏掉了。我们使用这种方式的好处就是我们永远只有一个  $P$ ，但是设置了两个通路。

## Explicit Congestion Notification (ECN) [ Floyd and Ramakrishnan 98]



- Traditional mechanism
  - packet drop as implicit congestion signal to end systems
  - TCP will slow down
- Works well for bulk data transfer
- Does not work well for delay sensitive applications
  - audio, WEB, telnet
- Explicit Congestion Notification (ECN)
  - borrow ideas from DECBit
  - use two bits in IP header
    - ECN-Capable Transport (ECT) bit set by sender
    - Congestion Experienced (CE) bit set by router

为了让发送端感知到路由器已经拥塞了，路由器是不是能和发送端进行一些沟通，这就是 ECN 来做的事情。这样就使得路由器和 TCP 发送端有了一种协议，使得可以协调一致地针对网络情况进行 slow down，但是增加了一个 signal 的开销。所以 ECN 就是针对传统方式，*implicit* 的方式（被丢包了 3 个 ack 或者 timeout）的问题的改进，显式地告诉 TCP 端。

它在 IP 包头里用了 2 个 bit 来显示已经拥塞了。router 往回发告诉 sender 你已经拥塞了，可以降低数据量了。router 之所以带宽利用率不能到 90% 以上，就是因为 router 可能还要做很多这种事情。

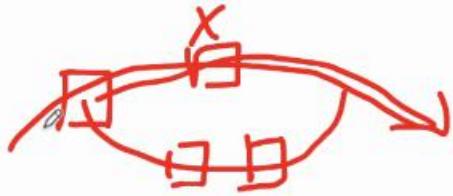
## Congestion Control Summary



- Architecture: end system detects congestion and slow down
- Starting point:
  - slow start/congestion avoidance
    - packet drop detected by retransmission timeout (RTO) as congestion signal
  - fast retransmission/fast recovery
    - packet drop detected by three duplicate acks
- Router support
  - RED: early signaling
  - ECN: explicit signaling

它的主要问题就是要能够感知网络中的拥塞。隐式地是通过 3 次 ack 和 timeout，TCP 的标准形式就是 slow start。向上探索。这里面还有一个重传机制，selective repeat 等。所以 timeout 造成的就是归一、duplicate acks 造成的就是减半。

路由器能做的事情就是 RED 和 ECN，它们的概念非常简单。RED 不需要和别人交互，ECN 就要和发送端能进行一些交互。可以告诉上游路由器不要走这里最近的通路（拥塞了），可以走别的路。



此时是不通知 TCP end point 的，可以在多 router 之间就做这个负载均衡的事情。上节课讲的 VLAN 就是 overlay network，它最重要的区别就是在物理的 LAN 上可以形成一个物理的组。

所以我们讲了那么多 RED, QoS, 主动、被动、显式、暗示等，大家对这些都要了解。

## QoS principle

如何提升网络中的服务质量，就是这堂课的核心。

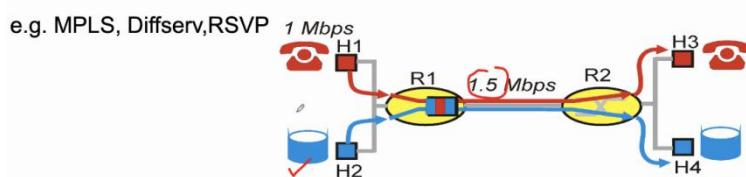
Improving QoS in IP Networks

- IETF groups are working on proposals to provide better QoS control in IP networks, i.e., going beyond best effort to provide some assurance for QoS
- Work in Progress includes RSVP, Differentiated Services, and Integrated Services
- Simple model for sharing and congestion studies:

传统的 best effort 我们认为是不好的，但是它也是一种 QoS。我们会将差分服务、资源预留等。

举一个简单例子，红色和蓝色就会竞争路上的资源。

- Consider a phone application at 1Mbps and an FTP application sharing a 1.5 Mbps link.
  - bursts of FTP can congest the router and cause audio packets to be dropped.
  - want to give priority to audio over FTP
- PRINCIPLE 1: Marking of packets is needed for router to distinguish between different classes; and new router policy to treat packets accordingly**



电话是有更高的优先级，那么如何提升给不同的应用给不同的优先级呢？IP 包头留了很

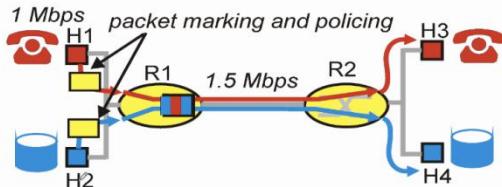
多位就是给 QoS 使用的。阿里使用 IPV6 的包头对不同的数据流类型进行了标注，通过高端路由器，使得它的云服务（叮咚买菜、支付宝），甚至是同一个数据流里的不同的数据包，进行了差分化的服务。

而在上例中是要识别并分出不同的任务。MPLS 在电信级别还是用得很多。DiffServ 就是有了 Level 之后怎么做分化的服务。RSVP 就是每个路由器针对每个 class 预留出充分的资源。比如每个路由器都预留的 1M 的资源，那么电话就会非常畅通，但是 RSVP 不好的地方就是可能会存在资源的浪费，我们不一定总是会使用到这 1M。这 1M 是充分保证服务质量的带宽，如果我们聊天之间有一分钟两个人都不说话，RSVP 是不能动态地调整带宽的。

## Principles for QoS Guarantees (more)

- Applications misbehave (audio sends packets at a rate higher than 1Mbps assumed above);
- **PRINCIPLE 2: provide protection (isolation) for one class from other classes**
- Require Policing Mechanisms to ensure sources adhere to bandwidth requirements; Marking and Policing need to be done at the edges:

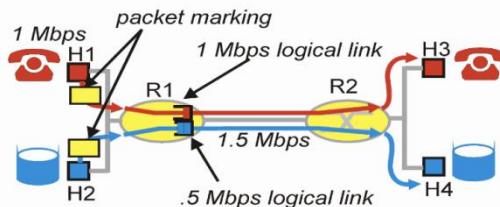
e.g. WFQ



我们要做一个 isolation，且高优先级的服务质量得到保障，且不能让低优先级的任务被饿死。轮询其实就是 WFQ 的基础。WFQ 就是说第一个人服务 3 秒，第二个人服务 1 秒，每个人占的比例就不太一样了。它的主要的特点就是时间隔离。Policing 就是协管，不能让应用的数据包一下子进来太多，比如电话业务乱发数据包，所以 Policing 就是不会让超出合理范围的数据流入网络，它就是一个看门的。这个东西一定是在边缘上做，所以才会有我们的 token marking。

## Principles for QoS Guarantees (more)

- Alternative to Marking and Policing: allocate a set portion of bandwidth to each application flow; can lead to inefficient use of bandwidth if one of the flows does not use its allocation
- **PRINCIPLE 3: While providing isolation, it is desirable to use resources as efficiently as possible**

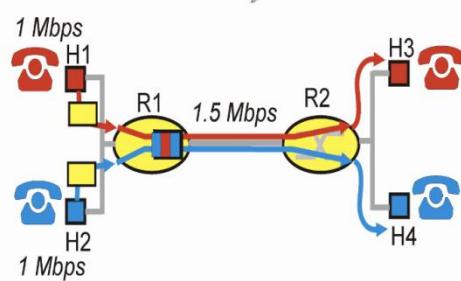


就是把一部分的带宽给一个应用。所以在带宽利用率上不一定是高效的，也就是打电话吵架不说话的例子。

## Principles for QoS Guarantees (more)



- Cannot support traffic beyond link capacity
- **PRINCIPLE 4: Need a Call Admission Process;**  
application flow declares its needs, network may block call if it cannot satisfy the needs

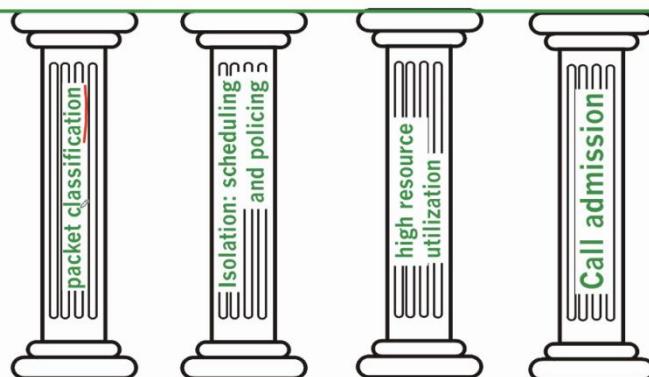


第四个 principle，要做 admission control，一次性不能进入太多，不能乱用网络带宽。  
不会让大量的数据包进入网络。

## Summary



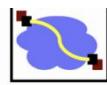
### QoS for networked applications



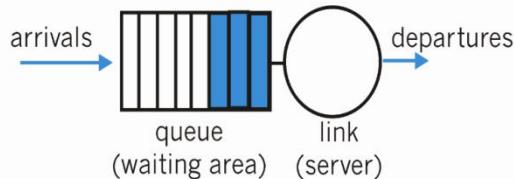
四大 policy，我们要在包里加一定的标志位。包里有很多的 reserved bit。另一个就是时间隔离和空间隔离。WFQ 就是时间隔离，多队列就是空间隔离。不能让多余的流量进来。

有了 principle 之后，我们后面就可以开始讲 policy 了。

## Scheduling And Policing Mechanisms



- **Scheduling**: choosing the next packet for transmission on a link can be done following a number of policies;
- FIFO: in order of arrival to the queue; packets that arrive to a full buffer are either discarded, or a discard policy is used to determine which packet to discard among the arrival and those already queued



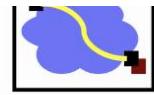
### Queuing Theory, Markov Chain

OS 里的调度和网络里的调度有区别。OS 理论上可以任何时间把一个任务停掉，可以通过上下文切换把它存起来。但是网络里一个包传输过程中是不能停的，96 个 0 就接收端认为你的包已经发完了，然后 checksum 一做就会出错。所以这叫做非抢占式的，而 OS 中优先级高的可以把优先级低的停住。

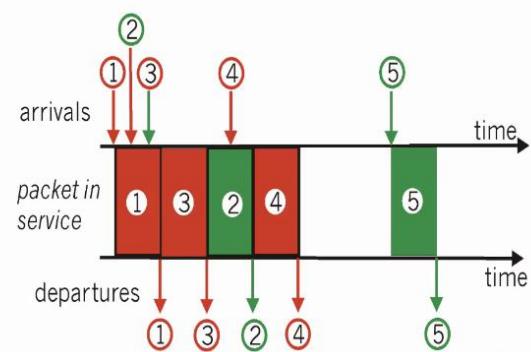
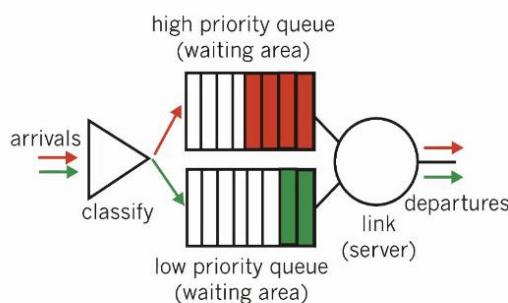
在可抢占的情况下，是有很好的理论基础，资源利用率在 70% 以下的时候，任务都是可以按时结束的。而网络中可能有一个很大的数据包和小的数据包，只要大的数据包一开始，一定会导致小的数据包没有赶上 DDL。

FIFO 就是最经典的调度方式，也是一种调度，更多的调度对象是 Queue 里的实体。一旦队列满了这种肯定不好。我们之前讲了运筹学里有 queue theory，还有马尔科夫链。

## Scheduling Policies



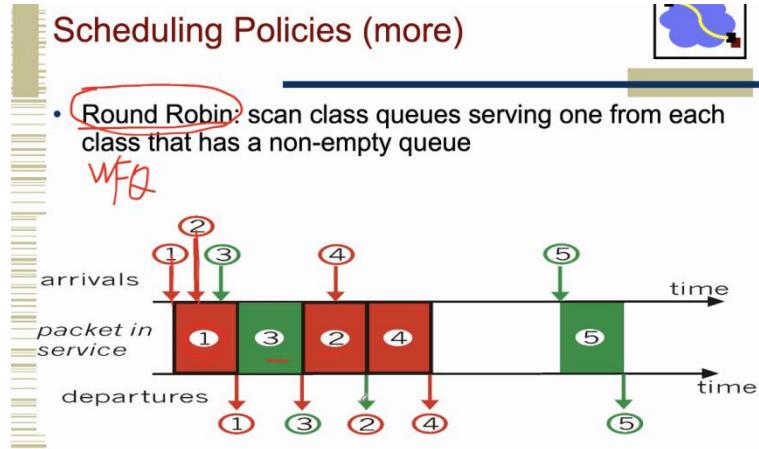
- **Priority Queuing**: classes have different priorities; class may depend on explicit marking or other header info, eg IP source or destination, TCP Port numbers, etc.
- Transmit a packet from the highest priority class with a non-empty queue
- Preemptive and non-preemptive versions



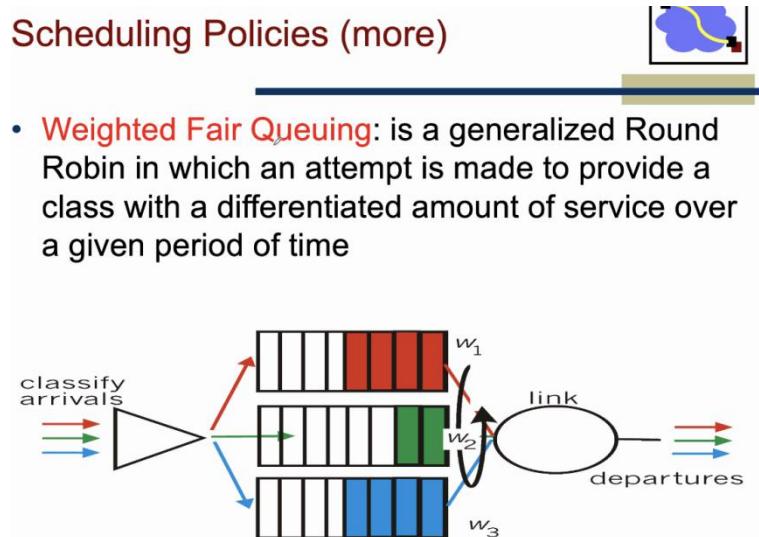
我们对不同的分类做了空间上的隔离，把高优先级的队列放到上面、低优先级的队列放

在下面，在网络里都是非抢占式的。数据包开始了就必须做到结束。

优先级队列就是一旦有高优先级的，link 就先发，然后再发低优先级的。假设高优先级队列永远不空，那么绿的就会被饿死。虽然 2 是比 3 先到的，但是进入了绿色队列。所以 1 之后先服务 3 再服务 2。直到 2 服务完了以后，即使绿色队列不空，也先服务红色队列。



这样就会相对来说比较公平。所以不同的调度方式对于同一个队列和 arrival flow 的输出是不一样的。

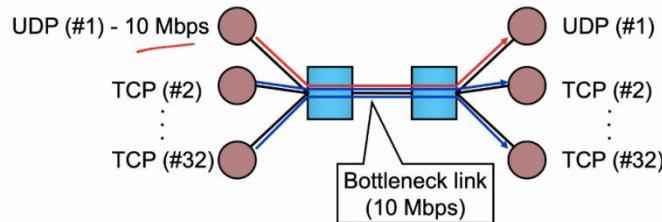


WFQ 就是每一个队列上都有一个权重。把包进行一个分组，分组怎么做？就是根据发送地址、接收地址等五元组参数进行分类，其实就是查包头进行分类。WFQ 就是 RR 的一种变种，只是每个权重不一样了。

## An Example

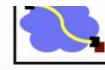


- 1 UDP (10 Mbps) and 31 TCPs sharing a 10 Mbps line



中间总体物理带宽只有 10M，那么怎么办？我们不可能让 UDP 全部占用了 10M，这些人既要公平共享，又要服务应用需求且保证高带宽利用率。路由器中的 ECN 和 ICMP 很多用的是 UDP 数据包，一般情况下这种控制级别的网络路由器之间协调的数据包的优先级会非常高。

## Policing Mechanisms



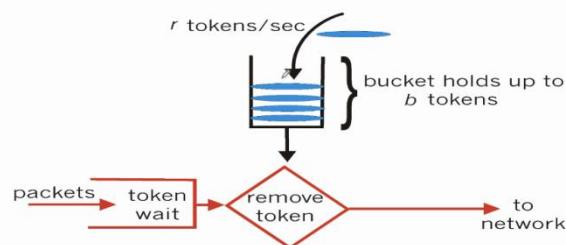
- Three criteria:
  - (Long term) **Average Rate** (100 packets per sec or 6000 packets per min??), crucial aspect is the interval length
  - **Peak Rate**: e.g., 6000 p per minute Avg and 1500 p per sec Peak
  - (Max.) **Burst Size**: Max. number of packets sent consecutively, i.e., over a short period of time

这里面就存在了 QoS 一定要精细化的问题，对于 Rate，它有很多种，比如 1 秒钟 100 个和 1 分钟 6000 个。这两个是不一样的，比如头 10 秒发了 5000，后 50 秒发了 1000，这样 peak 就会非常高。所以我们有 peak rate，还有突发的大小。

## Policing Mechanisms



- Token Bucket mechanism, provides a means for limiting input to specified Burst Size and Average Rate.



## Network Calculus Theory

前面我们已经讲过怎么做看门的事情了，这个就能解决电信里面，我们的手机、电脑、

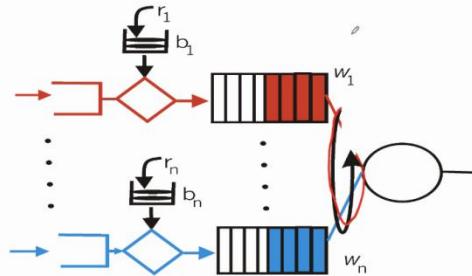
物理的网卡，带宽都是很高的，远远高于我们电信购买的带宽。运营商怎么控制我们的流量呢？比如买了 100M 的带宽，有 500M 的网卡可以多发数据包进去吗？

它会周期性地产生一些 token，token bucket 是有上限的，我们发送的数据包能进网络的前提是有 token 给到我们。最好最好的情况下，桶里有  $b$  个 token，当我们想发一个大数据流建立连接，我们至多拿到  $b$  个 token，所以 bucket 里通过  $r$  的速率产生 token，我们拿到 token 才能继续进到网络里。

## Policing Mechanisms (more)



- Bucket can hold  $b$  tokens; token are generated at a rate of  $r$  token/sec unless bucket is full of tokens.
- Over an interval of length  $t$ , the number of packets that are admitted is less than or equal to  $(r t + b)$ .
- Token bucket and WFQ can be combined to provide upper bound on delay.



我们之前再加一个 policing，即便我们使用 fix priority 也不存在饿死的情况，因为进来了这么多，因为被  $r$  和  $b$  这两个参数限制住了。这样空间隔离和时间隔离了。不同的 queue 在空间上隔离开了，我们只要做一些时间上的划分（RR,WFQ）即可。

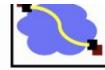
## Components of Integrated Services



- Type of commitment
  - What does the network promise?
- Packet scheduling
  - How does the network meet promises?
- Service interface
  - How does the application describe what it wants?
- Establishing the guarantee
  - How is the promise communicated to/from the network
  - How is admission of new applications controlled?

Integrated Serv 每个路由器在传输过程中要预留资源，连接建立过程中是比较长的，要所有路由器都预留好，服务质量才能保证。Packet Scheduling 是非抢占式的，我们还需要有一些 application 如何描述，其实就是在 TCP 层/IP 层打一些标记上去。所以 promise 可以在网络层建立一个很复杂的协议预留出来。

## Type of Commitments



- **Guaranteed service**
  - For **hard real-time** applications
  - Fixed guarantee, network meets commitment if clients send at agreed-upon rate
- **Predicted service**
  - For **delay-adaptive** applications
  - Two components
    - If conditions do not change, commit to current service
    - If conditions change, take steps to deliver consistent performance (help apps minimize playback delay)
  - Implicit assumption – network does not change much over time
- **Datagram/best effort service**

有一些 hard real-time application, 比如航空中也用了以太网技术做了保证。而我们互联网里用的都是 delay-adaptive application, 客户顶多会投诉。一个系统在运行中它的特性是否会随着时间变化而变化->时变系统，在网络中，我们的环境会发生很大的变化。比如我们组的虚拟机，网络服务提供商给我们的虚拟机和虚拟 CPU，都是会变化的。除非我们付很多钱让虚拟机绑定在一个物理地址上。

## Scheduling for Guaranteed Traffic



- Use **token bucket filter** to characterize traffic
  - Described by rate  $r$  and bucket depth  $b$
- Use **Weighted Fair-Queueing** at the routers
- Parekh's bound for worst case queuing delay =  $b/r$

token bucket filter 是专门用来看门的，对我们网络流入速度做约束。Parekh 是美国专门做网络的，它们研究了一辈子 network calculus。

## Token Bucket Characteristics



- On the long run, rate is limited to  $r$
- On the short run, a burst of size  $b$  can be sent
- Amount of traffic entering at interval  $T$  is bounded by:
  - $\text{Traffic} = b + r*T$
- Information useful to admission algorithm

从任何一个时间开始，最多  $b$  是满的，用完了  $b$  以后，只能以  $r$  的速率进网络。有了这个上界约束以后，就可以做保障性的服务。

## Guarantee Proven by Parekh



- Given:
  - Flow  $i$  shaped with token bucket and leaky bucket rate control (depth  $b$  and rate  $r$ )
  - Network nodes do WFQ
- Cumulative queuing delay  $D_i$  suffered by flow  $i$  has upper bound
  - $D_i < b/r$ , (where  $r$  may be much larger than average rate)
  - Assumes that  $r <$  link speed at any router
  - All sources limiting themselves to  $r$  will result in no network queuing
- [https://en.wikipedia.org/wiki/Network\\_calculus](https://en.wikipedia.org/wiki/Network_calculus)
- <https://leboudec.github.io/netcal/>

对于 network calculus，我们可以看一下 wiki。



### Contents

## An Introduction to Network Calculus

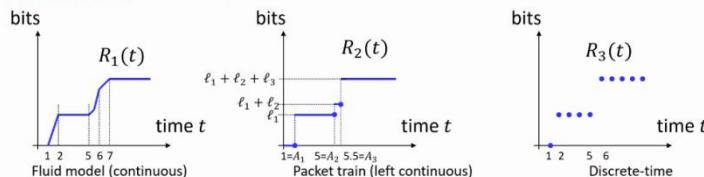
Jean-Yves Le Boudec  
IMAG / LIG  
Grenoble  
2019 November 12  
©

- Representation
- Arrival Curves
- Service Curves
- Packetizer
- Concatenation
- Shapers

我们只能简单地看一下他们在做什么，到达曲线和服务曲线。

### 1. Representation of Data Flow

Cumulative flow:  $R(t)$ , non-decreasing with  $R(0) = 0$

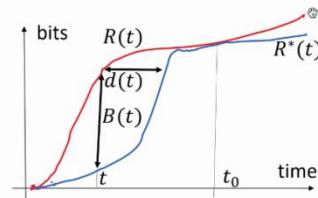


Daters:  $(A, L)$  with  $A = (A_1, A_2, \dots)$  (dates) and  $\ell = (\ell_1, \ell_2, \dots)$  (lengths in bits)

For a packet train:  $R(t) = \sum_{n \geq 1} \ell_n 1_{\{A_n < t\}}$

这是一个可堆积形式的 flow。它一定满足  $r \cdot b$  约束，也就是小于  $b + rt$ 。有连续形式的，也有 package 形式的，也有完全的离散化的。这种网络的数据流一定都是非递减的。

## Delay and Backlog



Backlog at time  $t = R(t) - R^*(t)$

If System preserves order for this flow: Delay  $\leq h(R, R^*)$

with  $h(R, R^*) = \sup_t d(t)$

and  $d(t) = \inf \{d \text{ s.t. } R(t) \leq R^*(t + d)\}$

(horizontal deviation)

红线就是到达曲线，蓝线就是服务曲线。我们可以认为线之间的水平距离就是 latency。我们只需要找到最大的那一点就是最大的延迟。我们只需要把两条曲线找出来，找到位置画竖线就是最大延迟。所谓的 backlog 就是存在 system 里的有多少个数据包。并且它认为网络里是一跳或多跳的。

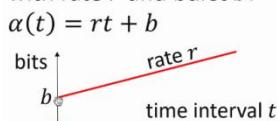
## 2. Arrival Curve

Flow with cumulative function  $R(t)$  has  $\alpha$  as (maximal) arrival curve if

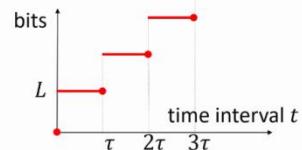
$R(t) - R(s) \leq \alpha(t - s)$  for any  $t \geq s \geq 0$

where  $\alpha$  is a monotonic nondecreasing function  $\mathbb{R}^+ \rightarrow [0, +\infty]$

**token bucket constraint ( $r, b$ )**      **periodic stream of packets of size**  
**(leaky bucket constraint)**       $\leq L: \alpha(t) = L \left[ \frac{t}{\tau} \right]$   
 with rate  $r$  and burst  $b$ :



[R. Cruz, PhD Dissertation 1987]



6

左图就是一个标准的到达曲线的 bound。如果下面的线随便画突然有一个抖动，那么在小时间范围内是违反 rb bound。右图是标准的周期性的数据流。还有一件事情是对于一个 flow 可以很容易地叠加起来。

## Aggregation Property

If every flow  $f$  has arrival curve  $\alpha_f$  then the aggregation

$$R = \sum_f R_f \text{ has arrival curve } \sum_f \alpha_f$$

If every flow  $f$  is token-bucket constrained  $(r_f, b_f)$  then the aggregation is token-bucket constrained  $(\sum_f r_f, \sum_f b_f)$

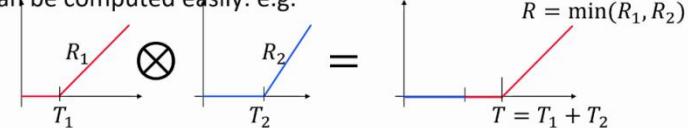
### Min-Plus Convolution of wide-sense increasing functions $[0; +\infty) \rightarrow [0; +\infty]$

$$f(t) = \inf_{s \geq 0} (f_1(s) + f_2(t - s))$$
$$f = f_1 \otimes f_2$$

This operation is called **min-plus convolution**. It has the same nice properties as usual convolution; e.g.

$$(f_1 \otimes f_2) \otimes f_3 = f_1 \otimes (f_2 \otimes f_3)$$
$$f_1 \otimes f_2 = f_2 \otimes f_1$$

It can be computed easily: e.g.



其中涉及到了 min-plus 卷积，所谓的卷积，这个和拉氏变换、傅里叶变换中是一样的。我们要把  $f_2$  反过来。

缺最后 15 分钟、

**2022/3/18**

## 了解 DPDK

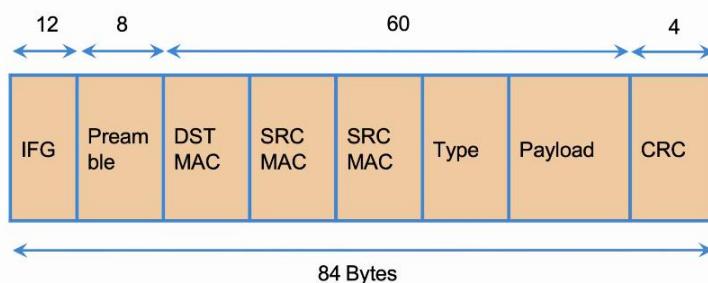
上堂课主要讲了 QoS，最主要的机制就是 FIFO, RED, ECN。DPDK 是用户态的协议栈，它现在是云平台高性能网络里事实上的标准，所有云商基本上用的是这一套，现在还衍生出了 SPDK，主要是用来做存储的。还有一个 PMDK，是主要来做 NVM 的，它是介于存储和 Disk 之间的，它是插在内存条上，但是性能比内存差一点、价格也便宜一点，它是非易失性的，但是性能接近内存。

### How fast SW has to work?

14.88 millions of 64 byte packets per second on 10G interface

1.8 GHz  $\rightarrow$  1 cycle = 0.55 ns

1 packet  $\rightarrow$  67.2 ns = 120 clock cycles



一个 10G 的网络，如果发 64Byte 的包，还要有 interframe gap + 8 Byte 的前缀，所以 payload 非常非常小。对于这种发，它可以发 14.88million 的 64 Packet 包。一个 CPU cycle 的时间间隔是 0.55s。所以 120 个 cycle 是很难处理从物理链接层到 IP 层到 TCP 层，一个 memory copy 和一个 interrupt 都是几百个 cycle，我们还需要解析、计算，这肯定是不够的。100G 接口，就只能有 12 个 CPU cycle。

## Comparative speed values

CPU to memory speed = 6-8 GBytes/s

PCI-Express x16 speed = 5 GBytes/s

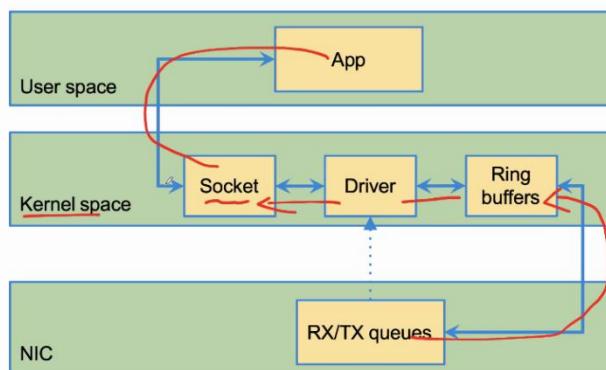
Access to RAM = 200 ns

Access to L3 cache = 4 ns

Context switch ~= 1000 ns (3.2 GHz)

上图也是一个 CPU 例子，这里的单位是 GBytes/s。理论上 CPU 到 memory 在 200~300G（注意 G 是 GBytes/s \* 8）。PCIE 现在有 3.0, 4.0 都在不断地研发，可以认为在 200G。后面两个数字也不太准确。

## Packet processing in Linux



就可以使用固定的接口来进行数据传输了。看代码是必须的，但是上述的一些性能参数是没有办法从代码里看出来的。这是大家在硕士要改进的事情，要理解人家为什么能想出来这种研究成果。现在看的都是 3~5 年内的研究成果。在找优化点之前，我们先要分析它的性能瓶颈。

## Linux kernel overhead

System calls

Context switching on blocking I/O

Data copying from kernel to user space

Interrupt handling in kernel

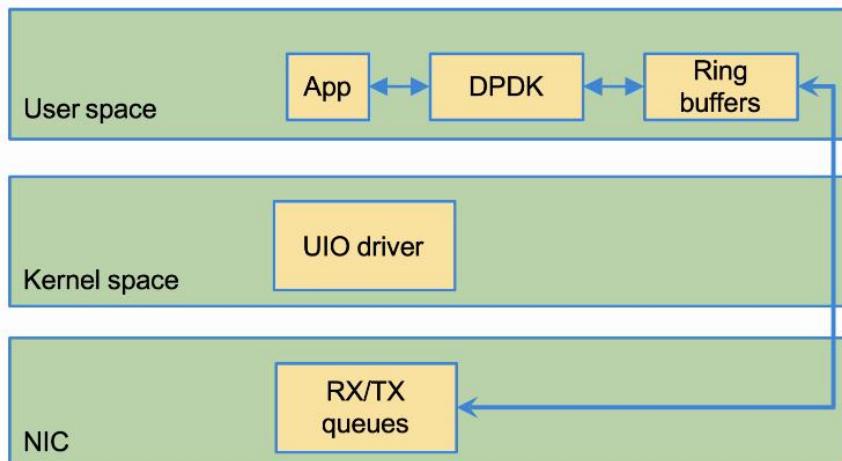
内核空间的 API 和用户空间的 API，功能是差不多的。性能为什么好，是代码看不出来的。还有一个就是 **blocking I/O**，发送端和接收端需要等数据回来再发。因为 **user space** 是没有办法在内核占用的内存上读数据的，所以我们需要一次数据拷贝。现在大家觉得内存空间的 **socket** 是有优势的，我们需要保留这个优势，但是也要降低这次数据拷贝的开销。然后还有一个中断处理，这在内核态也是需要几百个 cycle 的。

## Expense of sendto

Function	Activity	Time (ns)
sendto	system call	96
sosend_dgram	lock sock_buff, alloc mbuf, copy in	137
udp_output	UDP header setup	57
ip_output	route lookup, ip header setup	198
ether_output	MAC lookup, MAC header setup	162
ixgbe_xmit	device programming	220
<b>Total</b>		<b>950</b>

这就是 socket 发送一个数据包所使用的时间。单位使用 cycle 更好，cycle 和 ns 就是 CPU 的频率。这些是协议栈要做的事情，之前的上下文切换、内存拷贝都没有放在这里，因为一个上下文切换就需要 1000ns 了。

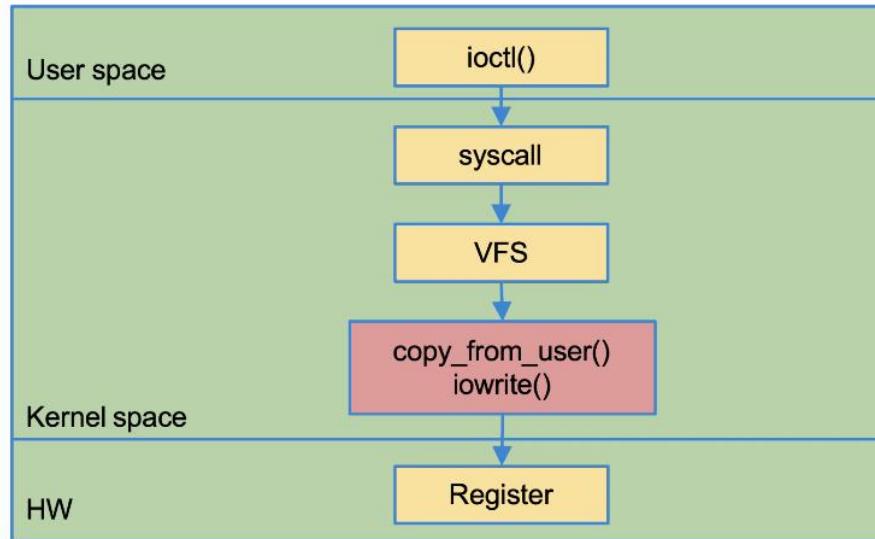
## Packet processing with DPDK



DPDK 的社区网站更新非常快。NIC 就是网卡，UIO 就是用户态的 IO 驱动，只做初期配置，整个过程中是绕过了内核，这样 NIC 通过 DMA 的方式把数据直接给到了用户态的内存中，这样就没有上下文切换和数据拷贝了。然后中断怎么来处理？它专门有一个 PMD, polling mode，它叫做轮询模式的 driver，用死循环（轮询）来替代中断。因为我们用 1 个 CPU 内核是处理不过来 10G 网络带宽的，

只要 memory 不空，就做协议栈的任务，我们可以简单理解为一个死循环。

# Updating a register in Linux

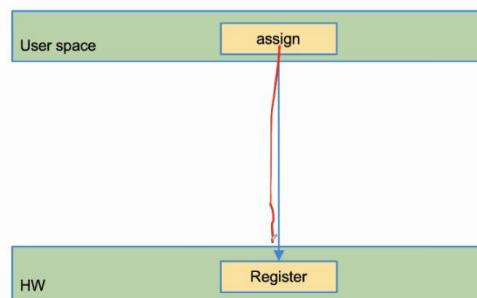


由于网卡是一个硬件（HW, hardware），大家还是要粗略理解网卡是怎么收发数据包的。会发一个终端上去，协议栈处理完了以后，告诉网卡“协议包已经处理完了”，让它发下一个数据包（中断）上来。网卡里也有一个小 buffer，如果我们不告诉网卡“上一个已经处理完了以后”，它就不会发数据包上来。Linux 里所有的设备都模拟成文件系统，通过 `iowrite` 写到网卡寄存器里。

我们传统 OS 是不让用户态直接写硬件寄存器的，但是如果高性能网卡没收到一个数据包都要通过这一长串路径写下来是很麻烦的，我们需要让用户态直接操作网卡寄存器，消除这个路径的开销。

技术上，不同的网卡设计的也不一样，同样是 PCIe 的设备，也需要寄存器的读写。但是如果性能够高，会同样面临类似的问题。

## Updating a register with DPDK



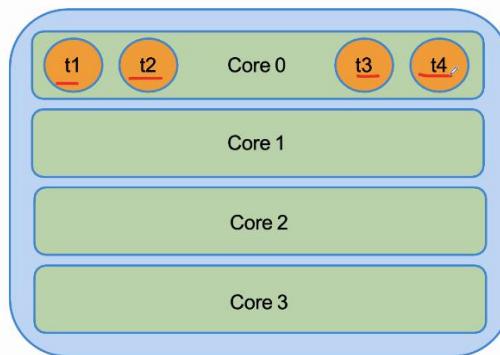
为了能让直通，我们需要费很大的力气，因为我们 OS 不能把用户态的这次尝试当做病毒给干掉。

## What is used inside DPDK?

- Processor affinity (separate cores)
- Huge pages (no swap, TLB)
- UIO (no copying from kernel)
- Polling (no interrupts overhead)
- Lockless synchronization (avoid waiting)
- Batch packets handling
- SSE, NUMA awareness

很重要是一个概念是亲和度（affinity），这和 NUMA 很相关。polling 主要解决 interrupt 的问题。

## Linux default scheduling



传统的情况下，线程可以在 Core 上乱跑，我们希望把线程绑死在某个核上。

## How to isolate a core for a process

To diagnose use **top**

“top” , press “f” , press “j”

Before boot use **isolcpus**

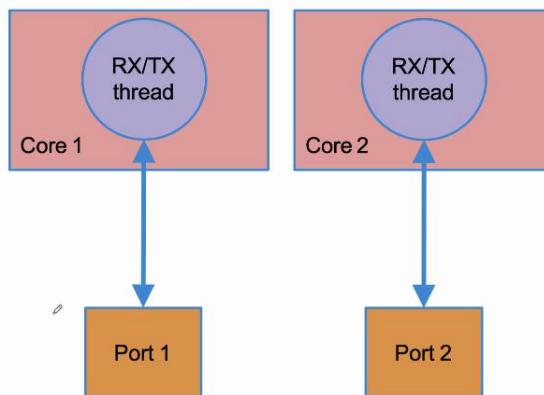
“isolcpus=2,4,6”

After boot - use **cpuset**

“cset shield -c 1-3”, “cset shield -k on”

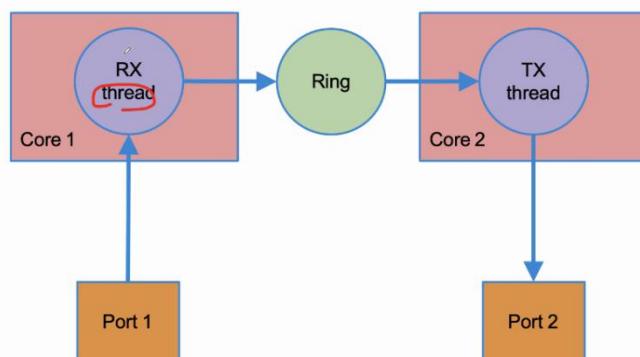
DPDK 就利用了这个隔离的机制。

## Run-to-completion model



这个就是通常所说的绑核。对于一些驱动程序经常会遇到这种情况，经常收包的 `thread`，我们要让它独占一个核。把其他线程的权限都关掉，这样 OS 就可以只给一个 `thread` 来用，因为如果有多个 `thread` 共享一个核，总归是时分复用的，有一个数据包过来了一个，总要受到 OS 调度器的开销，我们可能要等半天等现有任务执行完了再进行上下文切换。Run-to-completion 就是数据包过来了以后，一旦开始，就把所有的数据都处理完。

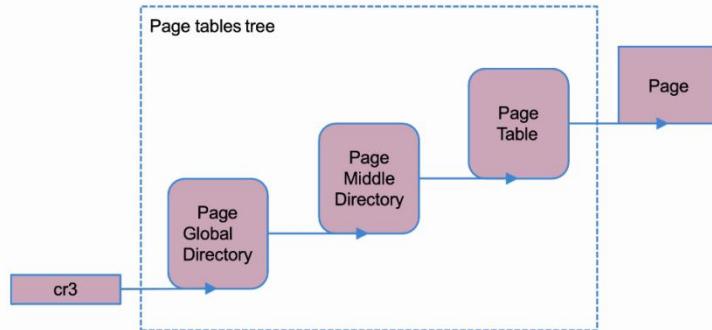
## Pipeline model



Pipeline 就是每个 `thread` 只做一个功能，比如第一个 `thread` 只做查表，第二个做别的事情，中间通过高效的方式互联，形成了一个流水线。每一个工人只做很简单的任务，但是吞吐量会很大。

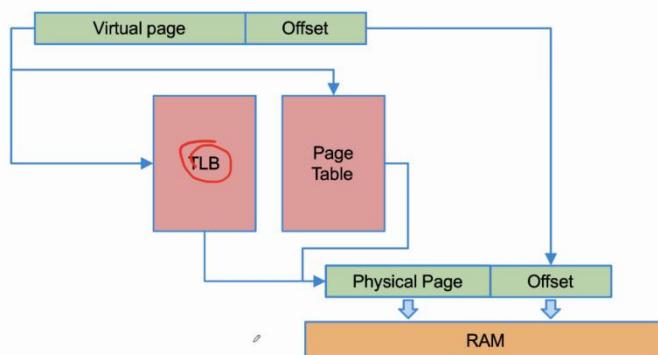
pipeline model 和 run-to-completion 在 DPDK 中都存在。

## Linux paging model



第一段找到第一级的，找到第一级以后再找到下一页。和 cache 是一回事，在虚拟化里有虚拟地址和物理地址的转换。

## TLB



## TLB characteristics

\$ cpuid | grep -i tlb  
size: 12–4,096 entries  
hit time: 0.5–1 clock cycle  
miss penalty: 10–100 clock cycles  
miss rate: 0.01–1%  
**It is very expensive resource!**

一旦 miss 了，延迟就会非常大。我们可以通过这个指令看一下 TLB 有多少个 entry。hit 和 miss 有一百倍的性能查表，我们需要降低 TLB 的 miss 率。

## Solution - Hugepages

**Benefit:** optimized TLB usage, no swap

Hugepage size = 2M

**Usage:**

mount hugetlbf /mnt/huge

mmap

**Library - libhugetlbf**

解决方案就是使用大页的方式，这样对应的条目可以提高五百多倍，这样 miss 就可以降低。有了大页以后，我们的核心思想就是让 TLB 管更大的内存空间，这样 miss 率就会少很多。尽量减少 swap。

## Lockless ring design

Writer can preempt writer and reader

Reader can not preempt writer

Reader and writer can work simultaneously on different cores

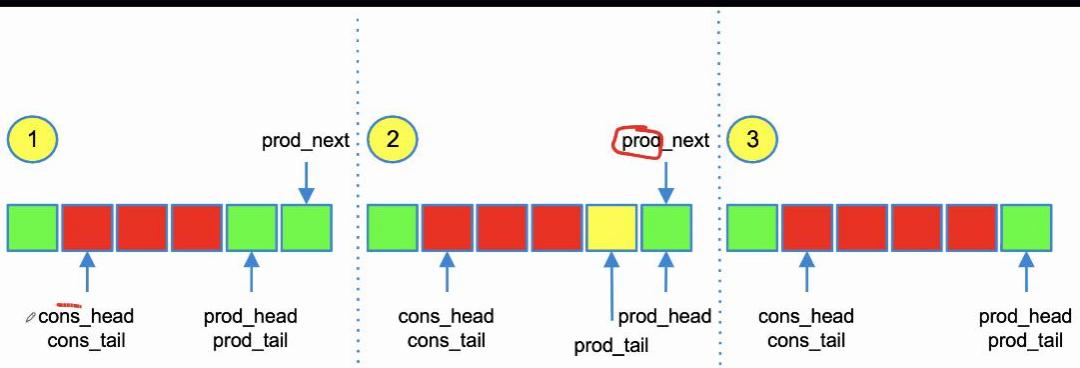
Barrier

CAS operation

Bulk queue/dequeue

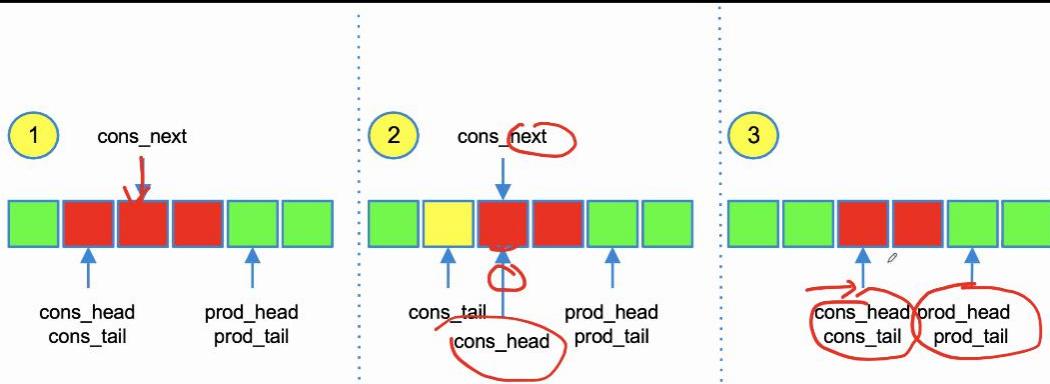
另外一个主要的知识点就是这么做无锁的 ring 和读写操作。让多核读写互不干扰同时进行。

## Lockless ring (Single Producer)



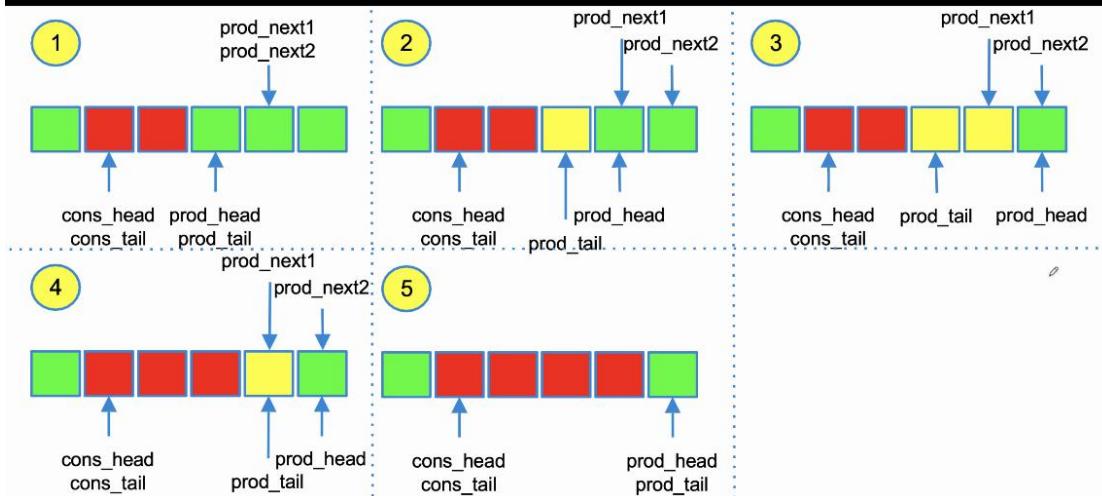
它会有一个 consumer，生产者有一个头和尾。写完了数据包以后，head 会读，所以整体从第五个变成了第六个，这就代表数据包读完了。

## Lockless ring (Single Consumer)



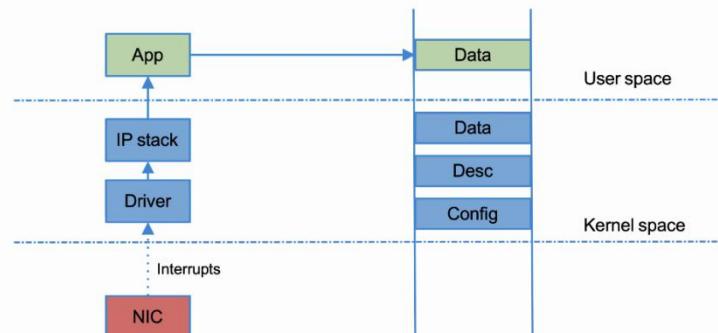
相差二就代表队列里有 2 个数据包。

## Lockless ring (Multiple Producers)



这里面讲了一个 multi-producers 的问题，也就是可扩展性。多核往队列里写的时候，就有 next1 和 next2，同时写两个地方。head 先挪两位，然后 tail 再移动。

## Kernel space network driver



所以内核空间实现和用户空间实现从代码上可能看不出什么区别。内核实现要先拿到描

述符，再把数据拷贝到用户空间里。它的好处就是对 APP 来说简单很多，直接调用 API 即可，但是就会造成一系列之前所提到的性能问题。所以，用户空间解决了什么问题呢？

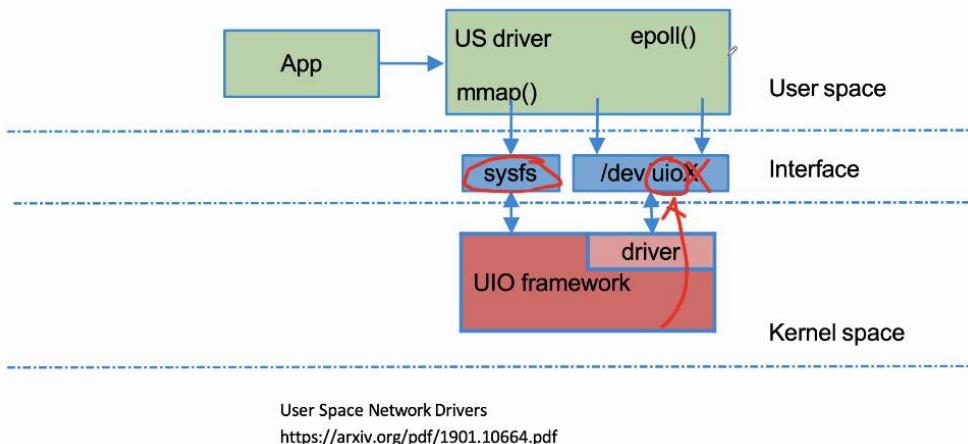
## UIO

*"The most important devices can't be handled in user space, including, but not limited to, network interfaces and block devices." - LDD3*

APP SSD

UIO 允许我们从用户态对 IO 进行访问。

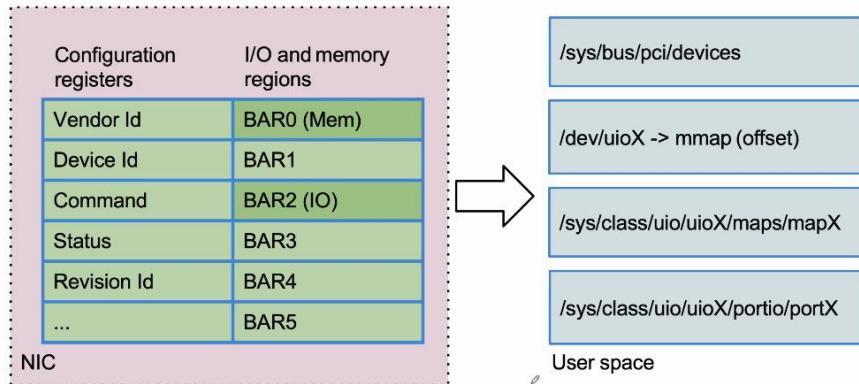
## UIO



User Space Network Drivers  
<https://arxiv.org/pdf/1901.10664.pdf>

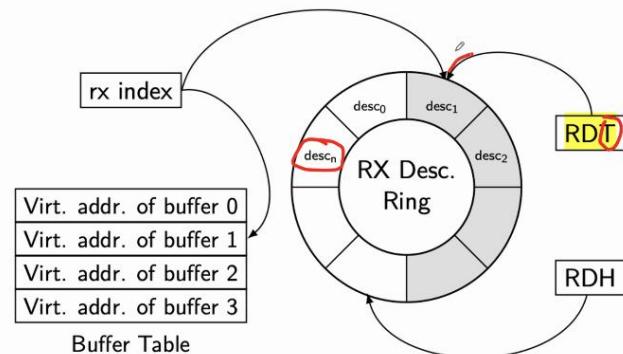
内核态注册完以后，用户态就可以对这几个注册的 IO 直接读写了，对网卡绕过内核直接进行操作。

## Access to device from user space

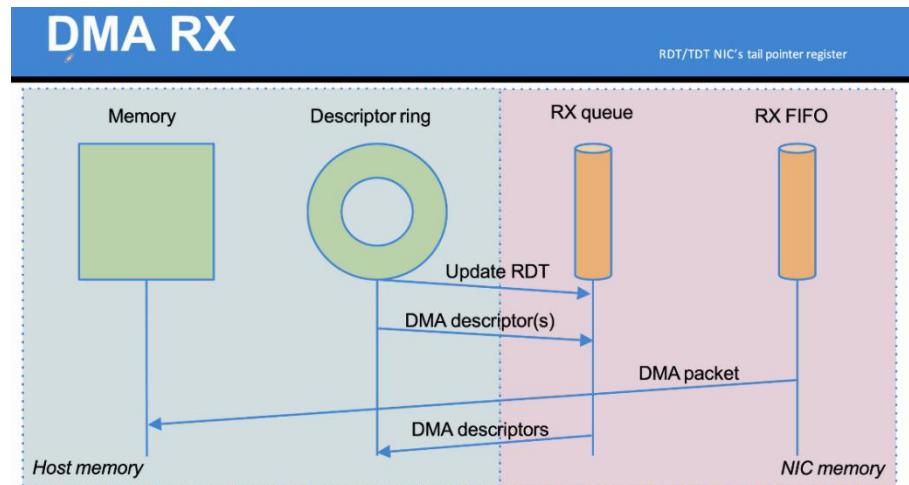


这里就是配置在什么地方。因为是 PCIE 设备，是有一个个 bar 的空间的。每个 Bar 里设计的就是里面的数据结构，要符合 PCIE 的标准，就可以自动地和 PCIE 进行交互了。

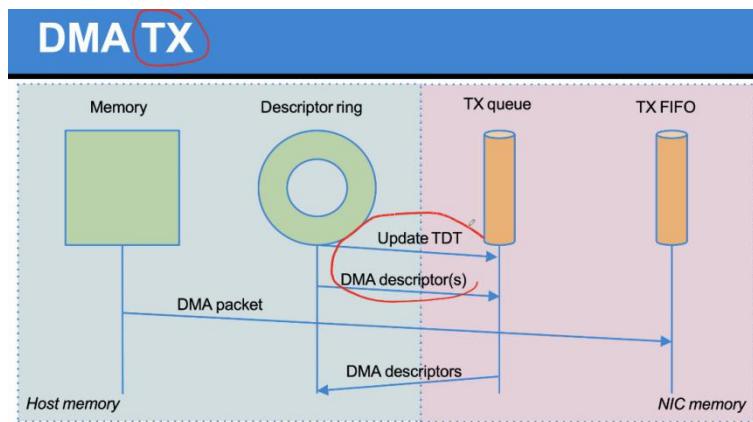
# Ring buffer



RDT 就是现在正在读的这个。它们的虚拟地址都在这，所以我们使用一个 descriptor 的队列就可以找到对应 buffer 的 receiver index 在哪里。

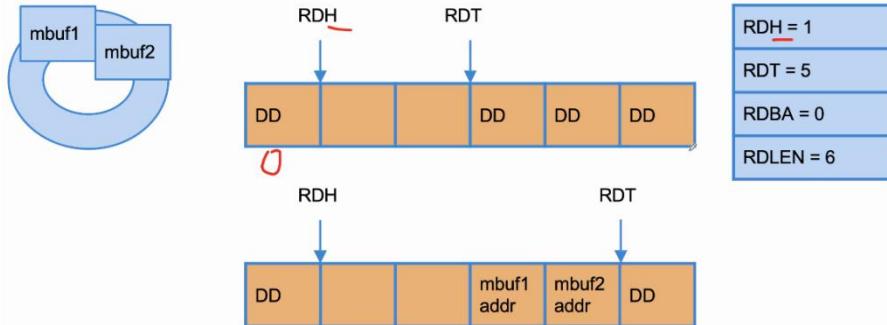


首先 RX 就是一个 receive。host 和 NIC 之间是通过 DMA 来操作的，之前有一个 config 和 descriptor，就是来配置 DMA 的。DMA descriptor 就知道了，写完以后还需要把 DMA descriptor 更新一下。因为我们要占一个地方。在不得已的情况下是可以不连续的。



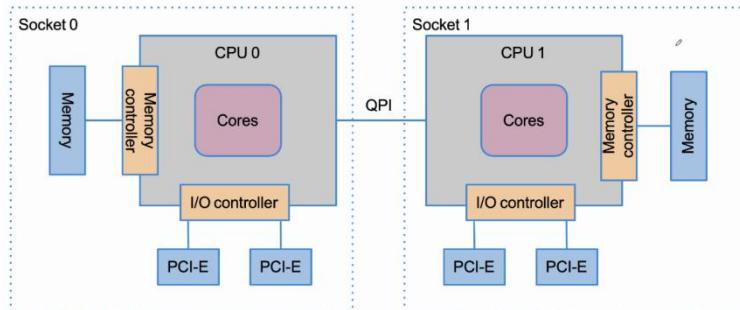
发送其实也是这两步。DMA 就是一个小的硬件，所以是双向的。不停地对 Descriptor Ring 进行配置。

## Receive from SW side



从第一个 DD 开始连续分了 6 个。RDT 从 3 挪到了 5。

## NUMA

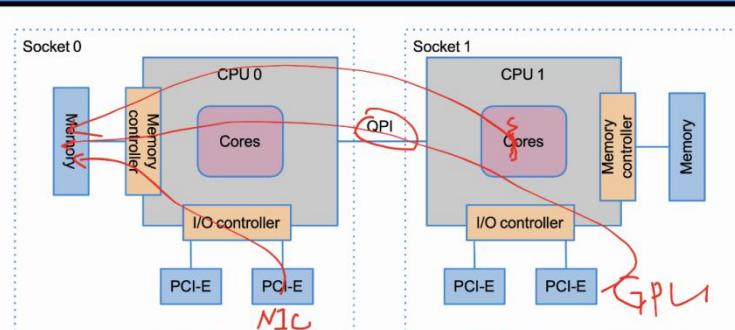


NUMA 就是我们的 server 是几路的。我们的台式机理论上只有一块 CPU，里面有多个核。但这个和核不是同一个概念。现在的 PC 机就是一块 CPU 插在主板的 socket 上。

CPU 之间通过 intel 的 QPI 进行连接。主板上的 chip set 其实就是负责联系 CPU 之间。假设 DMA 写在 socket0 的 memory，CPU1 尝试通过 memory 来读，开销就有 QPI + Memory controller。所以这就是 affinity（亲核性、空间局部性）。这种就是非同一模式（no uniform）的 memory access（NUMA）。如果我们再加个 GPU 和 FPGA，这样有大量的 PCI 设备要和 memory 进行交互，

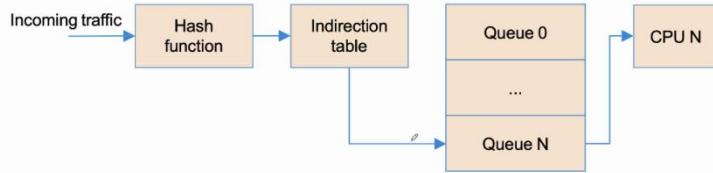
它们之间的相互位置我们假设如下，有一个应用的 thread 在 CPU1 上，网卡在 CPU0 上，memory 也在 CPU0 上，GPU 在 socket1。标注为 N1C 和 G1U。

## NUMA



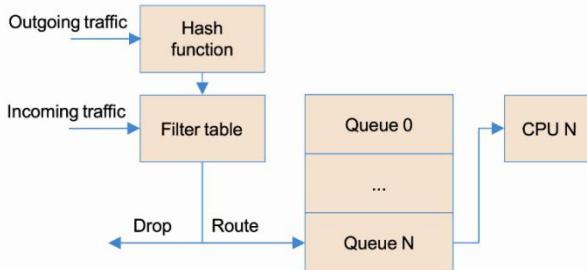
每次的 QPI 就会迅速变成系统瓶颈，怎么样把事情变得更好，这就是现在的一个课题。所以分布式系统的性能和理论上的吞吐量差了几个数量级。

## RSS (Receive Side Scaling)



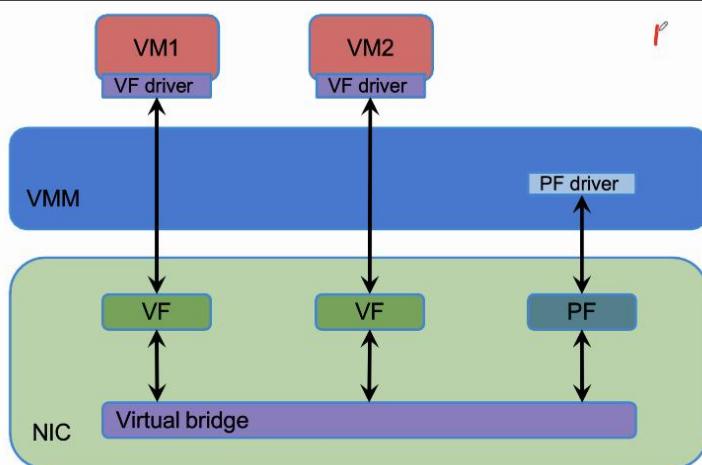
我们为什么 Queue 里可以多个人写，输入的 traffic 可以通过哈希找到这个位置，可以同时写几个位置。也可以多核地同时往外读包。对于网络数据包，发送端比接收端的开销要小。DMA 的通道有那么多个，所以一写就写进去了。

## Flow director



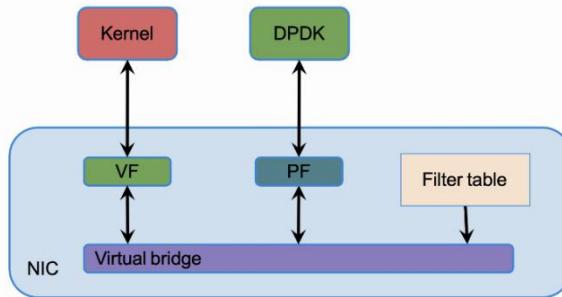
专门有一个迅速查表的加速器。match action table 就是 CPU 不参与，直接从内存写入。这个过程完全是用硬件来做。现在的网卡已经基础大量的硬件的，现在的智能网卡就是一个小 server 了，还有 FPGA, GPU 等。

## Virtualization - SR-IOV



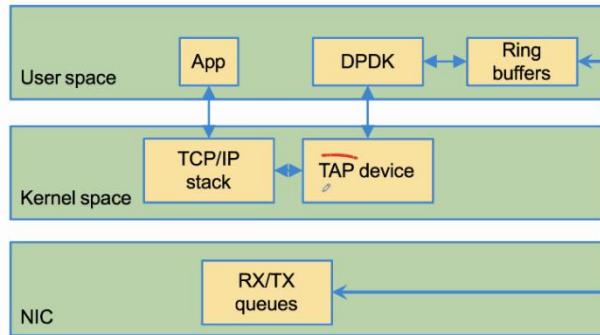
PF, VF 就是在做这件事情。有一堆 config。还有 L2 层的一个小交换机。它默认的是一个光口，我们能分发到多个虚拟设备，对上层的虚拟机来说，就认为拿到了一个标准的 PCIE 设备，所有生成都是硬件来做的，谁来配置呢？其实就是 VMM 来做。DPDK 在 server 上做非常方便，

## Slow path using bifurcated driver



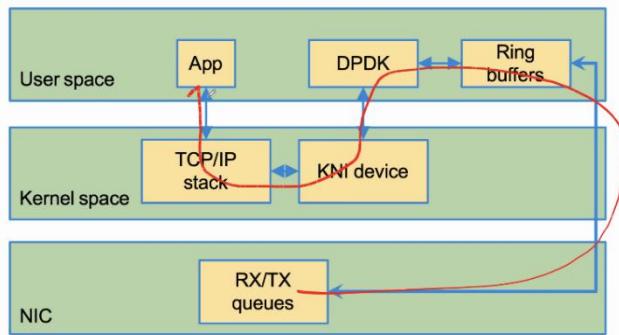
这是双 buffer 的一个 director。这两个是兼容的，比如 DPDK 也是兼容 PF 的。Flow vector 可以快速找到 destination。

## Slow path using TAP



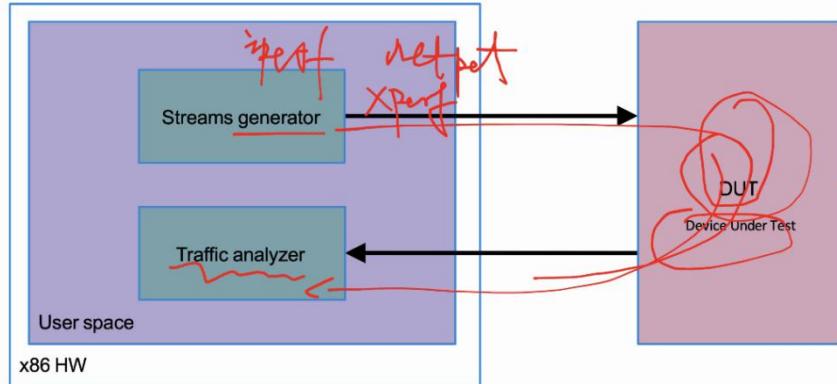
TAP 是一个最简单的输入输出设备，绕回了内核，所以很慢。我们的虚拟机，因为原始的 app 必须使用这个 interface，所以我们的桥接非常慢。好处就是兼容性高。

## Slow path using KNI



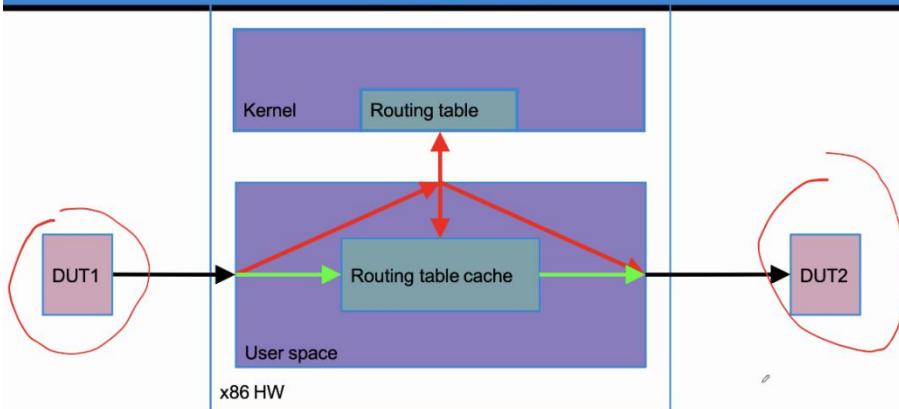
好处就是我们可以使用标准的 TCP/IP stack。如果是这样的话，最后一步是一定有数据拷贝的，但这一步也可以使用硬件加速器来优化。

## Application 1 - Traffic generator



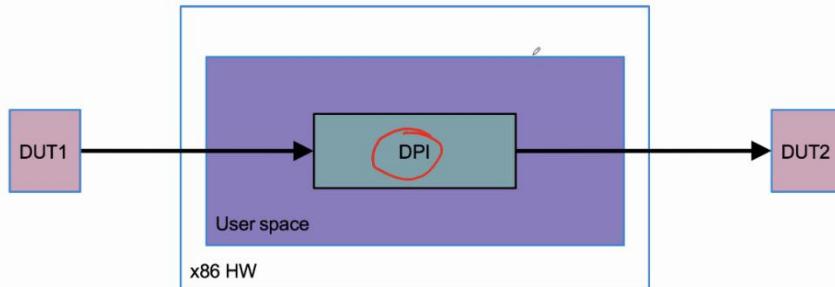
如果处理不过来，那么 analyzer 返回的信息就很少。这就是背对背测试。

## Application 2 - Router



还有一大类的测试，我们做的是一个 router，左边大量发数据包过来。中间有一定的硬件计算量，红色代表复杂的计算要进入控制面板，绿色的就是数据面。左边发了多少数据包，右边收到了数据包，我们就可以知道中间我们实现的这个路由器实现的好不好。

## Application 3 - Middlebox



这里有一个是 DPI，是做深度包检测。为什么我们发的数据会不让我们发，因为它会对明文做检测。现在的 DPI 也在不断地做演化中，还有 gateway、firewall 都叫 middlebox。

FD.io VPP，它是基于 DPDK 衍生出来的高性能网络框架。这个补充材料里讲的就会比较

细。中间还有一些新型的硬件做空间隔离。

**2022/3/22**

## Cryptography (密码学) and HTTPS

这是网络部分最后一次课。网络的安全和系统安全是有本质区别的，我们更多的强调的是数据包传输过程中的数据不能被篡改、不能被窃取，和机密计算还不太一样，后者更多的是内存中的数据不能被别人看到，以及 CPU 运行中的指针权限不能被截获。HTTP 主要是超文本图形语言，我们主要是讲 S 的内容。

### Flashback .. Internet design goals

- 1. Interconnection
- 2. Failure resilience
- 3. Multiple types of service
- 4. Variety of networks
- 5. Management of resources
- 6. Cost-effective
- 7. Low entry-cost
- 8. Accountability for resources

#### Where is security?

可计费是公司盈利的最重要的点。但是如果我们网络要是连不起来、总容易坏，那肯定不行的。但是这 8 点没有包含安全，计算机网络初期都是美国军方做一些大型计算来用的，所以它们对于网络节点数量没有那么多的预期，也没有考虑安全性。

### Why did they leave it out?

- Designed for connectivity
- Network designed with implicit trust
  - No “bad” guys
- Can’t security be provided at the edge?
  - Encryption, Authentication etc
  - End-to-end arguments in system design

它们认为国防部大楼内不会有恶意的人。所以现在我们要保证信息的机密性，以及对信息进行认证。**authentication** 其实我们信用卡上的磁条也是一种。所以我们要做端到端的，数据能够安全达到。

## What will we learn today?

- Why: brief history
- How: Cryptography and Steganography
  - Codes
  - Ciphers
    - Symmetric, Asymmetric
- Today: Kerberos, HTTPS

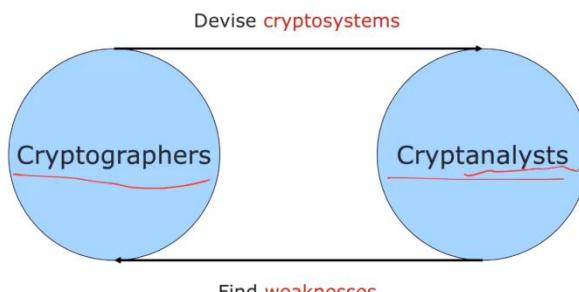
所以我们要从浅入深地讲一下。安全的历史比网络久的多，古代也有加解密的需求。对称加密和非对称加密我们更多介绍它的用法和方式。更多现在在云计算里，我们讲如何在云上安全传输数据并且分发密钥。密钥一旦丢了，那么就会出问题。

## A continuous arms race

- 1000's of years of **guarding secrets**
- Spartans – scytale, transposition cipher
- Romans – Caesar Cipher, rotation cipher
- Allied Analysis broke the ADFGVX
  - Led to the Zimmerman Letter decryption
  - Led to US involvement in WWI
- Breaking ENIGMA during WWII
  - Led to Allied tactical advantages

历史比较简单，所有的古代的军情都是要加密的。从斯巴达到罗马都有，比如 Caesar 加密。斯巴达的时候用的是一种移位转换。密码学的标准就是算法是公开的，但是自己的密钥要保管好。美国破解了日本的密钥，从而在太平洋战争具有优势。

### A continuous arms race



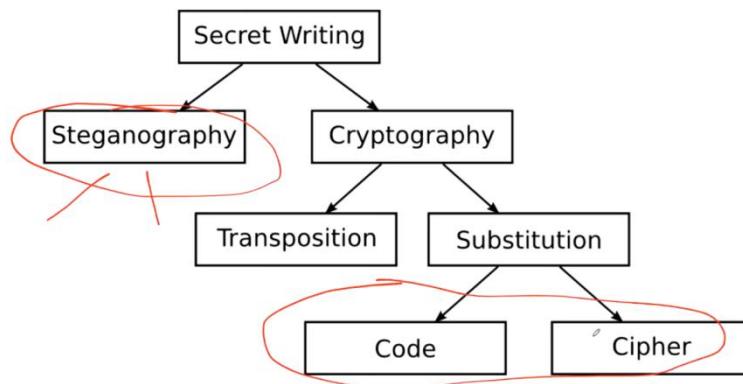
做加密算法的人和破解加密算法的人就是对抗的。

## Desired properties [Schneier96]

- **Confidentiality** – Ensure that an eavesdropper can not read a message.
- **Authentication** – It should be possible for the receiver of a message to ascertain its origin; an intruder should not be able to masquerade as someone else.
- **Integrity** – It should be possible for the receiver of a message to verify that it has not been modified in transit; an intruder should not be able to substitute a false message for a legitimate one.
- **Nonrepudiation** – A sender should not be able to falsely deny later that he sent a message.

1. 机密性：别人听到了也不知道是什么意思。
2. 认证性：需要知道信息的源头是谁。
3. 完整性：信息不能被改。
4. 不能抵赖：一旦发送了信息，这个消息就是你发送的。一旦发出，不能毁约。

## The history of communication



我们现在学的是密码学，主要是替换和移位。Steganohraphy 就是隐写术。

## Steganography

- The act of **hiding information**
- Often in plain sight...
- Example: slightly modify pixel data...
  - (R,G,B): (255,255,255) → (255,255,254)
- See app: **steghide**
  - Operates on both images and audio
  - Graph-theoretic basis
  - man steghide

其实就是把一个消息放在了一个可见的公用信息里，比如我们把图片改一位，这就是隐式地发送了一位 0 和 1 出去。

When successful, any eavesdropper **never knows**  
that a certain message has been transmitted.

而且信息是可以被抵赖的。

## Cryptography

- The act of **disguising information**
- Transforms what is called **plain text** into **cipher text**
- Two forms: transposition, and substitution
  - **Transposition scrambles** the plaintext letters
    - book → kobo
  - **Substitution replaces** words or characters
    - book → cjjl
    - Two forms: codes, and ciphers
    - **Codes replace words for other words**
      - book → bird
    - **Ciphers replace individual characters**
      - Title slide ciphertext: Gur Cbjre bs Xabjyrqtr

在密码学中，就是把明文变成密文，也就是移位和替代。黑话也是一种替代。

## The unbreakable cipher

- U.S. Patent 1,310,719
- **Vernam Cipher** – **one-time pad (OTP)**
- Mauborgne co-invented—thought of randomness
- **Shannon proved it is both unbreakable and fundamental!**
- Beautiful simplicity
- Incredibly powerful technology

在计算机里好用的就是 OTP(one-time pad)，这是美国的一个很重要的专利。香农从严格的理论角度证明了破解难度很大，基本上不可破解。

Is  $\oplus$  a good stream cipher?

Plain Text	Key	Cipher Text
0	0	0
0	1	1
1	0	1
1	1	0

在密码学里用的最多是异或，在不知道 key 的情况下，最终结果是 0 还是 1 完全推不出之前对应的情况。

## Vernam Cipher Encrypt

Plaintext	$\begin{array}{c} \text{"Hi"} \\ 1\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1 \\ \oplus\oplus\oplus\oplus\oplus\oplus\oplus\oplus\oplus\oplus\oplus\oplus\oplus \end{array}$
	Random OTP Key 1 1 1 0 1 0 0 1 0 0 1 1 0 1
	"tM"
Cipher Text	0 0 1 1 1 0 0 0 1 0 0 1 0 0
	"\x1c\$"

以上是用 OTP 对 hi 进行加密的方法，前提是解密方和加密方共用一个密钥。那么两边怎么共享一个密钥呢？两边怎么样在不安全的信道上共享一个机密信息（key）呢？

前面所说的方式已经很具体地展示了对称密钥的加密方法。

## Symmetric Key Cryptography

- Confidentiality via **shared keys**
- $E_k(M) = C$
- $D_k(C) = M$
- OTP is impractical because key length equals message length
- Alternatives
  - **Stream Ciphers:** RC4, A5/1,2,3 (GSM...)
  - **Block Ciphers:** AES, DES, Blowfish

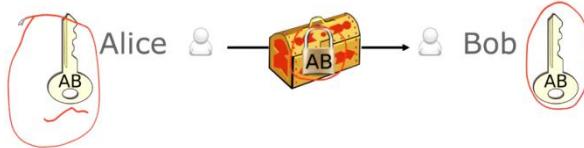
我们发现 OTP 密钥和信息一样长，但这很难实现。所以我们需要在一个流上不停地使用一个密钥，或者我们跨文件压缩一下。

## The treasure chest analogy



我们强调是数据传输过程中中间有一个人。

## The treasure chest analogy



只要锁和箱子够结实，我们就默认中间的信息是安全的。

## Hash Message Authentication Code (HMAC, MAC)

- Hash message using a hash keyed with shared key
- Produce MAC
- Alice or Bob verify integrity of messages based on these hashes

这是消息的认证。就是说，我们要对消息做类似于哈希的处理，这样我们就知道这个消息确实是某个人发送的。一旦被改动了，哈希值就不对了，我们就知道消息被改了。MD5也可以作为一个验证 integrity 的哈希值。

## Problem: Replay Attacks

- Eve can send messages again...with observed HMAC
- Fix: introducing nonces
  - Random bitstrings used only once
- Provides "sessions" for HMACs
- 

问题是消息被路由器劫持了，不停地发送之前的消息，所以我们要加一个序列号，只能发送同一个序列号一遍，replay 这个消息是无效的，这样就不会有重放攻击。

## Review: Symmetric

- Confidentiality – Stream/Block Ciphers
- Integrity – HMAC
- Authentication – HMAC and nonce

## Perfect crypto, what next?

- Yes, we have the technology
- But, we have a different problem
- **How can we share the one-time pads?**
- **Fundamental problem** in cryptography:

## Key Distribution

怎么让两边同时 share 一个 secret 呢？建立过程中被人窃取了（中间人攻击）会不会有问题？

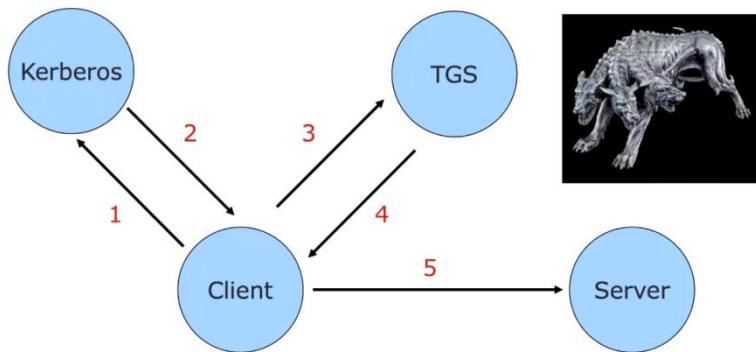
### Kerberos: Central Key DB

- **Key Distribution Center**
  - Database of clients and secret keys
  - Handles key distribution in symmetric case
- **Trusted Arbitrator Service**
  - Secure network authentication to servers etc.
  - Based on Needham-Schroeder's protocol
  - From MIT's Project Athena

这是一个很重要的技术，把消息给分发出去，也就是 KMS (key management system)。它其实就是一个数据库，相关的人来要才给出对应的数据。

### Kerberos: Authentication Steps

1. Request for ticket-granting ticket
2. Ticket-granting ticket
3. Request for server ticket
4. Server ticket
5. Request for service



它就是一个存放密钥的数据库，分发的是对称的密钥。它的优点就是可以维护密钥，只要服务器不被攻陷，我们的密钥就是安全的。但是不好的地方就是对于我们来说，Kerberos 本身可能成为被劫持的对象。

# Kerberos: The protocol

- c = client  
 s = server  
 K = key or session key  
 t = timestamp  
 V = time range  
 TGS = Ticket Granting Service  
 a = Net Address
- $K_c$  – one-way hash of client password
  - $T_{c,s} = s, \{c,a,v,K_{c,s}\}K_s$  – ticket
  - $A_{c,s} = \{c,t,key\}K_{c,s}$  – authenticator, session key optional
  - 
  - 1. Client to Kerberos:  $c, tgs$
  - 2. Kerberos to Client:  $\{K_{c,tgs}\}K_c, \{T_{c,tgs}\}K_{tgs}$
  - 3. Client to TGS:  $\{A_{c,s}\}K_{c,tgs}, \{T_{c,tgs}\}K_{tgs}$
  - 4. TGS to Client:  $\{K_{c,s}\}K_{c,tgs}, \{T_{c,s}\}K_s$
  - 5. Client to Server:  $\{A_{c,s}\}K_{c,s}, \{T_{c,s}\}K_s$

这里面是它做的一些事情。首先 client 发送一个 tgs 请求，kerberos 用 client 的密钥把两个信息加密，有一个是用 tgs 的密钥加密的 ticket，client 没有办法解密，但它可以继续向 tgs 发送 authentication。接下来 tgs 给 client 一个用 server 的 key 加密的信息，client 也不能解密。但是 client 就可以把这个消息发给 server，代表这个 client 已经收到了 tgs 的认可。

## Diffie Hellman Key Exchange [Wikipedia]

	Alice	Evil Eve	Bob
Step 1	Alice and Bob exchange a Prime (P) and a Generator (G) in clear text, such that $P > G$ and G is Primitive Root of P $G = 7, P = 11$	Evil Eve sees $G = 7, P = 11$	Alice and Bob exchange a Prime (P) and a Generator (G) in clear text, such that $P > G$ and G is Primitive Root of P $G = 7, P = 11$
	Alice generates a random number: $X_A$ $X_A = 6$ (Secret)		Bob generates a random number: $X_B$ $X_B = 9$ (Secret)
Step 2	$Y_A = G^{X_A} \pmod{P}$ $Y_A = 7^6 \pmod{11}$ $Y_A = 4$		$Y_B = G^{X_B} \pmod{P}$ $Y_B = 7^9 \pmod{11}$ $Y_B = 8$
Step 3	Alice receives $Y_B = 8$ in clear-text	Evil Eve sees $Y_A = 4, Y_B = 8$	Bob receives $Y_A = 4$ in clear-text
Step 4	Secret Key = $Y_B^{X_A} \pmod{P}$ Secret Key = $8^6 \pmod{11}$ Secret Key = 3		Secret Key = $Y_A^{X_B} \pmod{P}$ Secret Key = $4^9 \pmod{11}$ Secret Key = 3

还有其他密钥分发的机制，这个就是不经过第三方、也没有 DB 怎么做。这是大名鼎鼎的 DH 算法。Alice 和 Bob 自动生成  $P, G$ ，一般情况下是比较大的素数明文发送。两边生成随机数，不交换，这样 Eve 是不知道的。接下来我们计算  $Y_A$  和  $Y_B$  进行明文交换。最终这个密钥 3，Eve 是算不出来的。

## One-Way Functions

- Given  $x$ ,  $f(x)$  is trivial to compute
- Given  $f(x)$ ,  $x$  is hard to compute
- Example: increase entropy, break a plate
- Math: what we really want are trapdoor one-way functions

密文都是单向算法，也就是没有密文的情况下是算不出来的。

## Trapdoor One-Way Functions

- Given  $f(x)$  and  $y$ ,  $x$  is trivial to compute
- $y$  is some secret information
- Example: take apart a  $x = \text{watch}$ , pieces =  $f(x)$ ,  $y = \text{assembly instructions}$
- Math:  $16 * 24 = 384$ 
  - $x = 16$ ,  $f = *$ ,  $y = 24$

## Asymmetric Key Cryptography

- Confidentiality via private key
- $E_{\text{pub}}(M) = C$
- $D_{\text{priv}}(C) = M$
- Distribute public key, hide private key
- You made these with ssh-keygen -t rsa!
- Very practical, but generally slow
- Often (RSA, etc.) asymmetric methods are used to exchange symmetric keys for fast symmetric ciphers

解密的时候必须用私钥，这样公钥大家都有。但是用公钥加密以后，只有私钥才能解开。用私钥加密以后，所有有公钥的人就可以解开，这就是认证的过程，证明一定是有私钥的人发出来的消息。

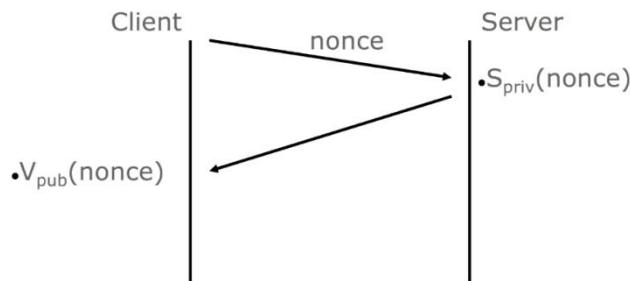
## Digital Signing

- $S_{\text{priv}}(M)$  – sign by encrypting (RSA)
- $V_{\text{pub}}(M)$  – verify via decrypting (RSA)
- Can sign entire messages
- But, often signing a hash is good enough
- Hashes are often shorter—quicker to compute

数字签名和 authentication 就是一回事，所有人都可以验证就是那个私钥签的名。

## Getting to Identity/Authenticity

- Send a **nonce**
- Used **only once!**



相当于服务器给 client 授权了，这样我们就拿到了 server 的授权。

## Review: Asymmetric

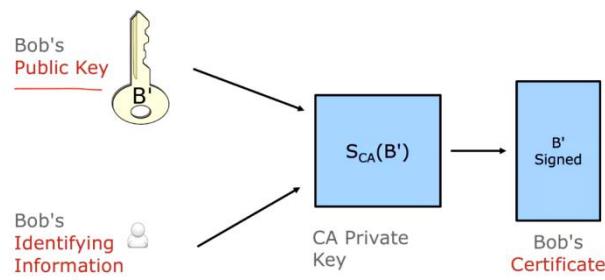
- **Confidentiality** – Public key encryption
- **Integrity** – Sign message with private key
- **Authentication** – Send a nonce challenge, use sign and verify

## Digital Certificates

- Issued to **prove identity**
- Requires trusted third parties
- We call these **certificate authorities**
- Or just trusted entities in a web of trust
- Used to implement TLS, HTTPS
- x.509 – standardizations

我们经常上网的时候看到钓鱼网站，证书就是防止钓鱼用的，我们需要有一个可信的第三方，从证书生成公钥，然后建立公钥和私钥连接，这样我们就可以保证持有私钥的人是可信的。在此之上，我们就可以建立起 TLS 和 HTTPS 的协议了。

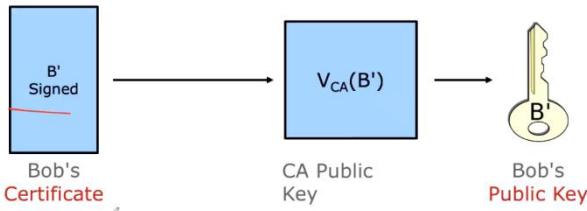
## Certificate Authorities: Issue



我们上传到证书机关之后，就生成了 Bob 的证书放在网上，大家通过这个证书来生成公钥。

## Certificate Authorities: Usage

Alice uses the CA's public key to verify Bob's identity and obtain a trustable public key for Bob.



## Public Key Infrastructure (PKI)

### • Certificate Authorities

- Bind public keys to certain entities ( $K_{B'}$  with Bob)
- DigiNotar – hacked, along with other CAs
- Admin Password: `Pr0d@dm1n`
- Iranian-based forged Google and more certificates

### • Web of Trust

- P2P model, let many others sign your public key
- Place trust in certain signatures
- GnuPG, PGP → implement this

在 P2P 中, 所有人之间每个人之间都有自己的私钥, 每个人都要知道对方是不是可信的, 在 PKI 就是 21 世纪最大的挑战。

Really? Yes!

**General Tab (Left Screenshot):**

Common Name (CN)	*.google.com
Organization (O)	Google Inc
Organizational Unit (OU)	<Not Part Of Certificate>
Serial Number	51:A9:99:AD:00:03:00:00:2E:74

**Issued By:**

Common Name (CN)	Google Internet Authority
Organization (O)	Google Inc
Organizational Unit (OU)	<Not Part Of Certificate>

**Validity Period:**

Issued On	8/11/11
Expires On	8/11/12

**Fingerprints:**

SHA-256 Fingerprint	63 80 03 73 A7 74 72 E9 3E TE 56 4E A2 17 2F C2 5c 37 05 71 BD 05 10 1C B4 3C 14 00 04 92 0F 64
SHA-1 Fingerprint	B9 93 D0 5C A0 7D 03 03 45 95 62 EC 18 1A EA BD 01 52 84 98 06

**Details Tab (Right Screenshot):**

- Certificate Hierarchy:** Shows the chain from **Builtin Object Token-Equifax Secure CA** down to **Google Internet Authority** and **google.com**.
- Certificate Fields:** Shows fields like Certificate, Version, Serial Number, Certificate Signature, Algorithm, and Issuer.
- Field Value:** A large text area for certificate content.

## HTTPS

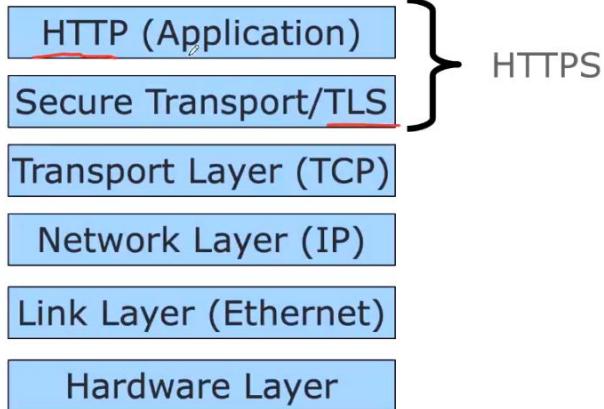
$$\underline{\text{HTTPS}} = \underline{\text{HTTP}} + \underline{\text{TLS}}$$

Netscape made SSL,

IETF made TLS

based  
on SSL

HTTP is unmodified!



Port 443 is dedicated  
for this.

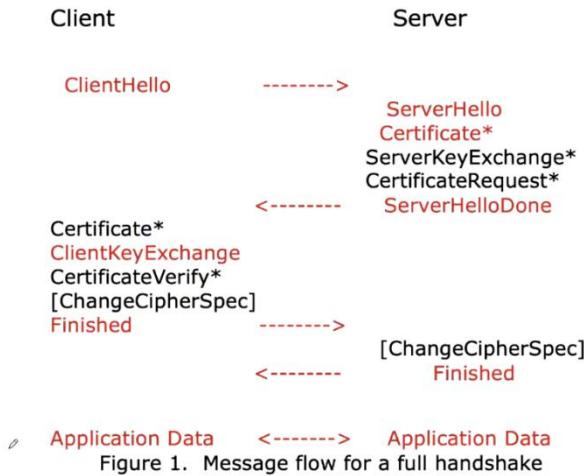
它就是通过 TLS 发送的超文本传输协议。

$$\text{TLS} - \underline{\text{RFC}} 2246$$

- Negotiate
- 1) Data **integrity** hash—HMACs
- 2) Symmetric-key cipher for **confidentiality** (DES, 3DES, AES)
- 3) Session **key establishment** (DH, RSA)
- 4) **Compression** algorithm\*
- HMACs and ciphers are **keyed in both directions**
- 6 keys needed total! All delivered with a **shared master secret**

RFC 成千上万，有对称的、非对称的。我们要建立起密钥的共享机制。在网络传输里，有时候会对 https 进行压缩，所以也有研究就是加解密和压缩/解压缩。

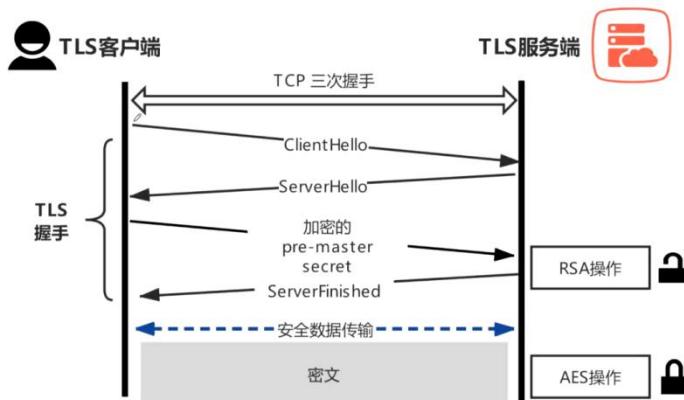
# TLS Handshaking [RFC 2246]



\* Indicates optional or situation-dependent messages that are not always sent.

在 TCP 建立连接以后，TLS 也要进行握手。Server 的 Hello 要验证证书，然后进行非对称的密钥交换。两边要建立一个连接，建立连接以后，我们需要把 spec 对应起来，如用了什么算法，要传输什么数据等。接下来再进入对称加密部分，传输真正要传输的数据。

## TLS Handshaking



## What's going on?

- Negotiation Hello's == protocols, crypto methods, compression
- Server certificate (signed public key)
  - Validate with browser set of CA's
- Client sends encrypted value to server, server decrypts proving private key ownership
- Secret value used to derive symmetric session keys for encryption and MACs

# Really? Yes!



Your connection to encrypted.google.com is encrypted with 128-bit encryption.

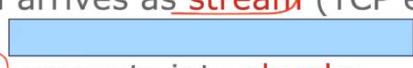
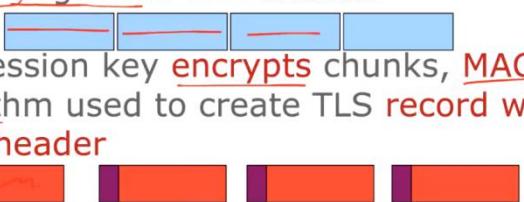
The connection uses TLS 1.0.

The connection is encrypted using RC4\_128, with SHA1 for message authentication and ECDHE\_RSA as the key exchange mechanism.

The connection is not compressed.

## TLS Data Stream

MD |

- 1) Data arrives as stream (TCP expected!)
- 2) TLS segments into chunks  

- 3\*) Session key encrypts chunks, MAC algorithm used to create TLS record with short header  

- 4) Records form byte stream for TCP layer  


## 2022/3/25(No)

现在看到的云通常会考虑公有云：阿里云、华为云、Google Cloud 等。现在使用云的时候，是租一台服务器，在云上创建一个实例。计算中心给每个同学创建了一个云的实例。背后的云可能就是一个很大的数据中心，数据中心里有很多很多机房、机架、机器。

### 为什么我们需要如此大规模的系统？

- 大量的对依赖大量资源的交互式服务出现，比如搜索引擎
- Web的崛起
  - 从数百万到数十亿的页面
  - 需要能够去索引这些大量的页面
  - 每天上亿对大量页面的搜索和查找
  - 极高的时延要求（亚秒级）

早期的计算机很多计算功能是做一些高性能计算，后来是一些交互式服务，比如 shell。有了搜索引擎之后，Google 是比较大的一个搜索引擎。现在的电商页面，每个页面中包含

了大量的链接。一方面我们要支持大量的 client，另一方面是大规模的存储。亚秒级是让用户没有那么难受，对于 12306 抢票，可能亚秒级就不够。这些需求促使着我们需要一个更大的系统，所以我们需要一个大型分布式系统。

## ► 早期的分布式系统



早期的时候 PC 电脑没有这么好，所以还会有工作站和 mini-computer 的概念。现在是 PC、笔记本、移动设备的概念，PC 往上是服务器的概念。当时伯克利以研究为目的建立了这样的一个分布式系统。

## 早期的分布式系统：独立地管理每个机器

```
for m in a7 a8 a9 a10 a12 a13 a14 a16 a17 a18  
a19 a20 a21 a22 a23 a24; do ssh -n $m "cd  
/root/google; for j in `seq $i ${$i+3}`; do  
j2=`printf %02d $j`; f=`echo '$files' | sed  
s/bucket00/bucket${j2}/g`; fgrun bin/buildindex  
$f; done" & i=${$i+4}; done
```

这个展示了当时它们管理机器的脚本，枚举一个机器，去做一些事情。但是枚举列表里我们发现 a11 和 a15 不在，这是因为它们挂了，每挂一台，我们需要手动修改这个脚本。

现在的云都是非常大规模的系统，可能机器数量是数以十万计。

## 大规模带来的挑战：亚马逊的案例

- 2005年左右，Amazon建造了大规模的数据中心，用来处理它的Web请求
  - 主要目的：支撑在线购物的业务场景
  - 核心目标：保证高可靠性



他们发现，整个数据中心没有他们想象的那么可靠（Reliable）

- 经常性过载（overloaded）或崩溃（crashed）

后面我们会看到亚马逊在云方面做出的工作。

## 大规模带来的挑战：谷歌的案例

~1 **network rewiring** (rolling ~5% of machines down over 2-day span)  
~20 **rack failures** (40-80 machines instantly disappear, 1-6 hours to get back)  
~5 **racks go wonky** (40-80 machines see 50% packetloss)  
~8 **network maintenances** (4 might cause ~30-min random connectivity losses)  
~12 **router reloads** (takes out DNS and external ips for a couple minutes)  
~3 **router failures** (have to immediately pull traffic for an hour)  
~dozens of minor 30-second blips for DNS  
~1000 **individual machine failures**  
~thousands of hard drive failures  
slow disks, bad memory, misconfigured machines, flaky machines, etc.  
Long distance links: wild dogs, sharks, dead horses, drunken hunters, etc.

有一些是软件和用户使用错误的情况。这里其实就有两个认知。

1. 可靠性不能完全靠硬件来管了，以前的硬件能力比较强，比如硬盘有 checksum，但是现在规模太大，硬件搞不定，所以需要软件来支撑可靠性。

2. 机器的利用率平时不是特别高。如果我们预留了很多资源，我们可以轻松地处理峰值，但是平时我们的设备就浪费了，没有办法去用它。中小企业和个人很难维护大量的机器。

在这种情况下，我们就有云计算的业务了，平时的机器闲着也是闲着，亚马逊就开启一个租赁服务。这就是云的业务模型雏形。大公司建立一个非常大的数据中心，就可以抵充数据中心的建造成本和维护成本。小公司和个人缴纳租金。

这里面就需要软件的可靠性，以及基础设施来支持租赁服务。这就是云 OS 的雏形，需要支持云计算的业务模式。

## 云与云操作系统

- **操作系统的目標：下接硬件，上接应用**

- **云操作系统：**

- 下接云上硬件、底层操作系统等
- 上接云原生应用



传统意义上的 OS 就是对下接硬件，对上接应用。现在的 OS 的概念是广义的概念，应用不太管以下是什么东西的，比如对做 ML 的人来说，TensorFlow 也可以认为是 OS 的一部分，因为它不需要去修改 TensorFlow。对云 OS，它下接的是云上硬件和底层 OS，它管理的是分布式系统中的很多机器，需要对应云上的机器和单机的 OS，对上的就是云原生（为云开发的应用）的应用和一些其他兼容云的应用。

## 云操作系统的演化经过了大量研究和实践

- 一个核心的思路：

– *Provide Higher-Level View Than “Large Collection of Individual Machines”*  
---- Jeff Dean



05年的时候，亚马逊就用分布式系统来提供云服务的支持了。核心思路就是：做一层封装把下面的机器封装起来，每一台机器就是一个实例。

### 云OS的需求一：怎么抽象计算？

- 小明现在有一大群集群和设备
- 怎么把这些能够计算节点包装起来，让云租户可以用呢？



什么样的计算抽象呢？

早期可能就是提供一个已经配好的物理机，只是按照租金收费。

- 计算抽象的一些需求（来自老王，小明朋友）：

- 不同的租户会有不同的环境需求：Linux或Windows? LAMP?
- 单个租户的不想租一个完整的机器，它只想租部分的资源



什么样的计算抽象呢？

Linux? Windows?

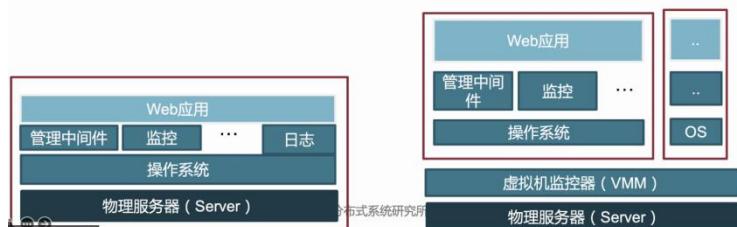


老王

大家有不同的环境需求。

### 计算实例：虚拟机 ( Virtual Machines )

- 每个租户能够拥有一个完整的“机器”
  - 包括自己独立的OS、存储设备、网络设备
- 一个物理机上可以通过虚拟机监控器创建多个VM



解决方案就是提供虚拟机，这样我们就可以在虚拟机里配置自己的操作系统，不用的时候它就是硬盘上的一个镜像，这样一个物理机上就可以通过虚拟机监控器创建多个虚拟机。这样就是虚拟机提供了一个便利的抽象。

## 云OS的需求二：资源共享和性能隔离

- 在不同的任务或者租户之间共享资源的机制十分重要

- 混合部署不同类型的任务：提升资源利用率
- 但是，会导致不可预测的性能抖动等



这里的需求主要是说，在租户很多的时候，随着云服务逐渐发展，越来越多人去组机器，租户可能有不同的需求，比如电商有 APP/网站交互性的需求（响应及时）、推荐系统的需求（把一些用户的购买行为放在一起做机器学习分析，这种用法需要大规模计算，属于 batch 分析的需求）。我们可能混合部署这些需求，可以提升资源利用率。

- 在保证（性能）隔离的同时提供共享

- Memory Ballooning：让虚拟机之间共享内存
- Linux Container（是不是说namespace和cgroup更合理？）
- ...

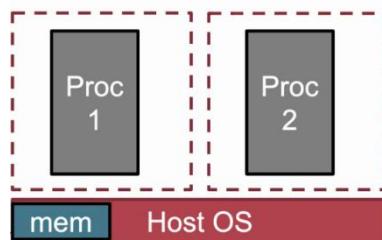
所以我们还需要保证性能上的隔离，这和安全上的隔离略有不同，这里说的是我们的性能基本上是稳定可预测的。Ballooning 可以让虚拟机之间共享内存。Linux Container 是在 namespace 和 cgroup 这两个抽象上建立的抽象，这个抽象就可以提供比较强的性能隔离和安全隔离。

## 计算实例（2）：容器（Container）

- 容器（Container）

- 和虚拟机类似，同样能够给用户一个完整的运行环境
- 容器之间没有私有的OS，而是共享一个OS内核
- Linux Namespace提供安全隔离，cgroup提供资源隔离（OS课程会介绍）

大家只需要知道容器利用了这两个概念提供了隔离。我们可以认为容器是第二代虚拟化技术，为什么它的虚拟机之间跨代了呢？它可以认为是一个轻量化的虚拟机，因为下面共享了一个操作系统，同时还是提供了一个虚拟环境。



这种情况下，内存管理更简单，我们只需要使用 mmap 来管理。我们可以通过 unmap 归还给 Host OS，在 mmap 到其他容器中。很多出于安全考虑，公有云使用虚拟机作为外壳，里面可能还是使用容器。

这门课的目标就是把大家学到的东西综合起来，编程、软件工程、数据库、网络的东西、编译器的东西。这门课就是给系统软件的同学做一个收口，把我们学到的东西做一个 Project，还有 CSE 的内容。这是大家融会贯通的一件事情。主要会按照 project 的计划。

我们刚才讲了两种计算实例：虚拟机和容器。

### 云OS的需求三：怎么管理大量的计算实例？

- 虚拟机和容器的技术初步满足了小明的需求
- 小明又遇到了新的问题：实例太多了
- 一个物理机跑10个VM，1000个物理机有10000个VM



这两个出来了以后突破了物理机的限制，可能就有数量级增加的虚拟机。

### 集群调度

- 虚拟机和容器往往比物理机还多得多
- 集群调度目标：将容器和VM放在特定的物理机上
  - 处理资源需求、限制等
  - 在单个机器上高效地运行多个计算实例
  - 处理机器的错误 ( Failure )
- 经典编排系统
  - Kubernetes
  - Apache Mesos



所以现在就有一个集群调度/编排的概念。在这样一个大规模的环境里，就会有一个编排的系统。我们需要找到错的并且重启起来。

### 云操作系统：现在长什么样子？



## 懒惰的开发者

- 小明发现，开发者（租户）时常抱怨
  - “这个云平台太难用了，要自己租多台机器进行分布式计算，要管好多东西，我只想简单的写应用！”
- 用户真是太懒了！



早期的云平台是不太好用的，虽然有图形化界面，但是管机器依旧很麻烦。虚拟机是一个很泛化的需求，它可以跑很多东西，但是不好用。高层应用框架可以支持特定需求的。

## 云OS的需求四：应用框架

- 老王建议小明在云OS上面支持一个应用框架
- 更高层的应用框架：
  - 抽象简单
  - 能够支持特定需求，如大规模并行计算
  - 弹性好
  - 等

框架也具有弹性，框架需要支持不同的用户，可定制化是吸引不同用户的方法，但是会带来一些潜在的复杂性。

## 经典框架：MapReduce

- 简单的`map()`、`reduce()`抽象
- 拥有一个好的应用框架的基本要素：
  - 隐藏了大量细节：局部性、调度、容错、slow machine、等等
  - 易用：在大规模计算中十分容易使用
- Hadoop: MapReduce的开源版本
- MapReduce也经常被用在分布式、数据中心里
- 后面我们会看到更加贴近“云原生”的框架

MapReduce 最开始是大规模计算的框架。

## 云操作系统：现在长什么样子？（2）



## 现在的云计算

- **公有云与私有云**
  - 公有云：提供云服务给任意开发者
  - 私有云：通常在内部（如公司内部等）提供云服务
- **按需 (On-demand) 提供计算资源**
  - 通过大量简单易用的API
  - 充分利用大量的大规模数据中心和基础设施
- **云计算正在成为基础能力，类似水电**

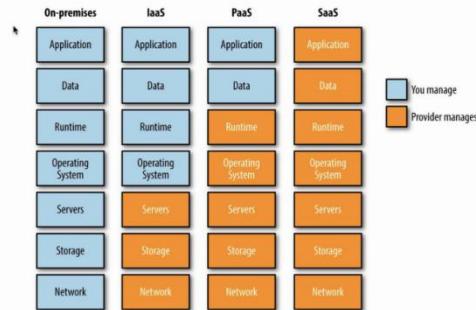
百度云一开始是做云存储的，所以按需提供了计算资源、网络资源、存储资源。当然云是分不同类型的，现在看到的更多的是公有云，还有私有云的概念。比如交大网络中心提供的服务就可以类似于私有云，它是不对外提供的。

## 云服务商 (Cloud Service Providers)

- **一些例子：**
  - 亚马逊：EC2 (2006)
  - Google: AppEngine (2005), 其他服务 (2008)
  - Microsoft: Azure (2008)
  - 阿里云、腾讯云、华为云、Ucloud等等
- **大量的用户将自己的应用迁移到云平台**
- **云平台的API、服务内容等也在急剧演化**

# 云计算：我们现在在哪里？

- 从IaaS到SaaS（当然，你可以听说过很多其他的XaaS）



云现在有不同的方式，提供裸金属（物理机）的情况也是存在的。当时亚马逊提供裸金属的情况是面向有技术的客户的，现在为了支持多样化的云用户需求，也提供了这种产品。上图从左到右越来越“傻瓜化”，云服务商来管理更多的事情。PaaS 是把 OS 和运行时（Python, Java v8）装好了，用户只需要提供程序和数据即可。SaaS 提供的就是一个云服务，比如语音识别服务、识别服务。现在已经有预训练的模型部署好了，只需要提供请求调用接口就行了。

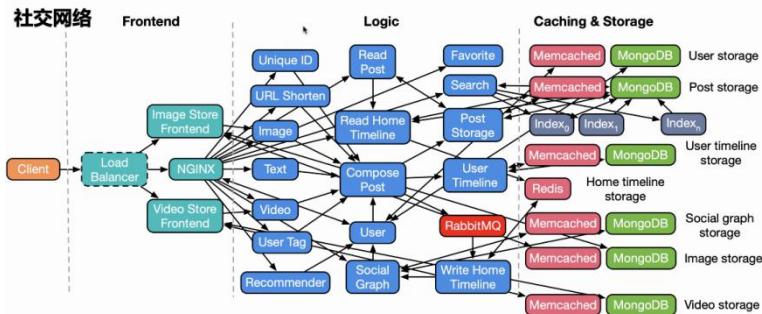
# 云计算：我们现在在哪里？

## • 云原生（Cloud Native）

- 新的应用场景：微服务、Serverless
- 新的设备：快速设备RDMA、GPU等
- 新的云基础组件：API网格、服务网格、监控日志等等

后面我们会介绍云原生的概念和新需求。

# 新的应用场景：微服务



传统是 moralless 的 web 服务。大家在 web 课程中搭建的更多是大规模的服务。现在我们会把每个需求裁剪成微服务。这些可能都会有自动扩展的需求。

9:20

2022/3/29

这门课没有期末考，以 project 为主。李老师那里 1 学分，这里 3 学分。这节课先讲一下 lab 的安排。

# 云操作系统 Lab：Minik8s

## 概述

- **迷你容器编排工具：minik8s**
  - 在多机上对满足CRI/OCI接口的容器进行管理
  - 实现Kubernetes中Pod, Service等抽象（mini版）
  - 支持容错、自动扩容等高级功能，支持GPU应用
  - 基于minik8s平台，实现microservice或serverless的自选平台搭建
  - 强烈建议参考kubernetes的实现方式
- **高度自由的小组作业**
  - 不对实现方式、编程语言等做限制，鼓励利用etcd, zookeeper等现有组件
  - 锻炼小组协作、项目管理等软件工程技能，过程中需提交文档并管理GIT项目
  - 以答辩为主要验收方式，演示实现的功能，按照功能判分

K8s 的简化版，希望在多机上实现 CRI/OCI 的管理。我们希望我们的 miniK8s 支持 pod 和 service 等抽象，支持容错、自动扩容。大家可以选择 microservice 或者 serverless。希望大家参考现有的 K8s 实现。不限制任何实现方式，希望大家使用 etcd、zookeeper 现有的组件作为我们的系统的一部分，减少重复造轮子。接下来我们讲一下实验的功能

## 基本功能要求：实现Pod抽象

- **Minik8s需要支持Pod抽象**
  - 通过yaml对Pod进行定义和配置
  - 基于Pod抽象管理容器的生命周期
  - Pod内需要运行多个容器，通过 localhost可以互相访问
  - 可以通过get pod, describe pod之类的指令获得Pod的运行状态等信息

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
cert-manager	cert-manager-6d68bf57f-2x49w	1/1	Running	3	27d
cert-manager	cert-manager-cainjector-64c949654c-7vxkh	1/1	Running	4	27d
cert-manager	cert-manager-webhook-6b57b9b886-b5lpq	1/1	Running	2	27d
default	annotation-second-scheduler	1/1	Running	2	26d
knative-serving	activator-769bf8bbc6-c4n2d	1/1	Running	2	26d
knative-serving	autoscaler-6d488889d8-8bqzl	1/1	Running	2	26d
knative-serving	controller-864dd5d5c8-tkpcl	1/1	Running	2	26d

POD 是 K8s 最基本的单元。miniK8s 也要实现 pod 的抽象，支持 yaml 对 pod 进行配置。此外 miniK8s 支持基本抽象，需要支持在一个 pod 里运行多个容器。

## 基本功能要求：实现Service抽象

- Minik8s需要支持Service抽象

- 对Pod的访问应当通过Service进行
- 对外提供Service的虚拟ip，能够通过虚拟ip访问Service，由minik8s将具体请求转发至对应的Pod
- 可以通过get svc之类的指令获得service的信息

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
activator-service	ClusterIP	10.96.84.154	<none>	9090/TCP,8008/TCP,80/TCP,81/TCP	27d
autoscaler	ClusterIP	10.96.120.127	<none>	9090/TCP,8008/TCP,8080/TCP	27d
autoscaler-bucket-00-of-01	ClusterIP	10.96.228.248	<none>	8080/TCP	27d
controller	ClusterIP	10.96.36.103	<none>	9090/TCP,8008/TCP	27d

外面的人要访问到这个资源，就需要 service 抽象。

## 基本功能要求：实现ReplicaSet/Deployment抽象

- Minik8s需要支持ReplicaSet或者Deployment抽象

- 对Pod指定多个replica并监控状态
- Pod发生crash或者被kill，可以自动启动pod重新加入service
- 通过Service的请求以一定负载均衡策略分配到各个Pod处

```
kind: Deployment
metadata:
  name: nginx-deployment
labels:
  app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	0/3	0	0	1s

接下来我们是要实现 deployment 的抽象。一个 service 在底层可以用多个 pod 来支撑。

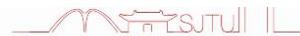
## 基本功能要求：水平自动扩容 (Horizontal Pod Autoscaling)

- Minik8s可以根据任务负载等对Service中Pod的replica数量进行动态扩缩容
  - 需要对Service下的Pod实际资源使用进行定期监控，监控对象包括至少两种资源类型，其中CPU为必选项，剩余的可以是memory, IO等
  - 用户可以指定动态扩容配置，至少包括扩容的目标，minReplicas, maxReplicas, metrics
  - Minik8s需要能够配置扩缩容的策略，策略包括两个部分：
    - 何时进行扩缩容
    - 扩缩容如何进行，即扩缩容的速度是怎样的

有了 replicaset 抽象以后，我们接下来就可以通过对数量进行调整来进行扩容或者缩容

的效果。

## 基本功能要求：DNS，容错



- DNS

- 通过yaml配置文件对Service的域名进行配置
- 支持同一个域名下的多个子路径 path对应多个Service

- 容错

- 为Minik8s的控制面实现容错
- Minik8s的控制面发生crash, 不影响已有Pod的运行
- Minik8s的控制面重启后, 已部署的Pod与Service均可以重新正常访问

不同的子路径可以对应到对应的 service。

## 基本功能要求：支持GPU应用



- 本课程将为同学们提供交我算平台的访问能力

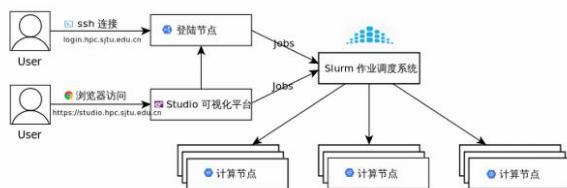
- 交我算平台通过Slurm工作负载管理器来调度任务
- 能够运行诸如CUDA程序的GPU应用

- Minik8s能够帮助用户运行GPU应用

- 类似Kubernetes中Job抽象
- 用户提交GPU应用+运行脚本
- Minik8s利用交我算编译运行

- 演示需要实现简单CUDA应用

- 实现矩阵乘法和矩阵加法即可, 需要利用硬件的并发能力



交我算目前使用 Slurm 工作负载进行调度, 用户可以把任务提交给交我算平台, 然后交我算就会调度到计算节点上。Minik8s 要提供 K8s 类似 job 的抽象, miniK8s 接入交我算后端平台。

## 基本功能要求：多机Minik8s



- 最终需要在多机上实现容器编排的功能

- 支持Node抽象, 支持新节点加入集群
- 支持Scheduler调度Pod。Pod在启动时, 首先根据scheduler的调度策略, 将Pod分配到适合的node上运行。
- 需要实现scheduler的调度策略
- Service的抽象应该隐藏Pod的具体运行位置
- Deployment和自动扩容均可跨多机

我们 service 访问只需要访问 service, 而不知道 pod 具体运行在哪里。

## 可选功能

### 可选功能：Microservice



- 基于minik8s实现简单的Service Mesh（参照Istio）
  - 对Pod流量进行拦截：在Pod基础上以实现网络代理，拦截所有进出Pod的流量
  - 支持自动化服务发现：Service Mesh利用minik8s提供的API，自动发现部署中所有Service和Pod，并告知每个Pod中的网络代理，使得被劫持后的网络流量仍然能够按照minik8s的定义正常分发
  - 支持高级流量控制功能，包括：
    - 灰度发布：按照配比分配流量；按照正则表达式匹配结果分配流量
    - 滚动升级：在Service可以在不停机的情况下完成对内部Pod的升级过程
  - 自行实现简单的microservice应用，或部署开源的microservice应用以展示上述功能

我们需要对进出 Pod 的所有流量进行拦截，接下来做灰度发布或者滚动升级。我们需要部署一个 microservice 应用来演示上述功能。

### 可选功能：Serverless



- 基于minik8s实现简单的Serverless平台（参照OpenWhisk或Knative）
  - 支持Function抽象，用户可以通过单个文件（zip包或代码文件）定义函数内容，上传给minik8s，通过http trigger调用函数，函数需要至少支持Python语言。
  - 支持Workflow抽象。用户可以定义Serverless DAG，包括调用链和分支
  - 支持Scale-to-0：Serverless实例在函数请求首次到来时被创建，并且长时间没有函数请求再次到来时被删除（Scale-to-0）。
  - 支持请求级别的函数扩容：Serverless能够监控请求数，当请求数增多时，根据相应Policy自动扩容至>1实例
  - 实现Serverless应用（Workflow）以展示上述内容

## 分组作业

### • 三人小组

- 自由分组，三人一队，指定组长和组员
- 分组信息提交至canvas，每组提交一份即可
- 周五（4月1日）前未分到组的同学，我们将随机分配

## 阶段性考核

- 该Lab分多次迭代完成
  - 开题时指定迭代计划，按照迭代计划完成目标
- 阶段性材料提交与考核答辩
  - 开题：提交开题报告，内容包括人员分工，选定的可选题目，迭代计划等
  - 过程答辩（中期）：主要是介绍进度，对流程进度进行评价，并提交中期报告
  - 验收答辩：对所有完成的功能进行演示，并提交验收报告
- 软件工程要求：标准流程开发
  - CI/CD（测试必须涵盖验收时演示的功能）
  - Git分支管理（要求创建私有Gitee项目进行管理，0抄袭！）
  - 统一代码风格
  - .....

## 阶段性考核

### • 时间节点

- 开题报告截止日期：4月8日，开题报告要求见Lab文档
- 中期答辩日期：5月6日（暂定）
- 验收答辩日期：第16周（暂定）

## 云操作系统：Minik8s Lab

本次大作业需要同学们完成一个迷你的容器编排工具minik8s，能够在多机上对满足CRI接口的容器进行管理，支持容器生命周期管理、动态伸缩、自动扩容等基本功能，并且基于minik8s实现两个自选要求，自选要求包括微服务、和Serverless平台集成等。实现过程允许、鼓励同学们使用etcd、zookeeper等现有框架。

### 基本功能要求

#### 1. 实现Pod抽象，对容器生命周期进行管理

Minik8s需要支持pod抽象，可以通过yaml来对pod进行配置和启动。

基于pod抽象，Minik8s应该能够根据用户指令对pod的生命周期进行管理，包括控制pod的启动和终止。

pod内需要能运行多个容器，它们可以通过localhost互相访问。

用户能够通过pod的yaml配置文件指定容器的参数，包括：

- kind: 即配置类型，应该为pod
- name: pod名称
- 容器镜像名和版本
- 容器命令 (command)
- 容器资源用量 (如1cpu, 128MB内存)
- volume: 共享卷
- port: 容器暴露的端口

yaml的格式可以自行设计，包含上述内容即可，可以自行修改或添加新字段和新内容。建议参考kubernetes的写法进行设计。

Minik8s可以通过get pod, describe pod之类的指令获得pod的运行状态，展示的运行状态信息和kubernetes类似（包括pod名，运行时间、运行状态等）。可以同时运行多个pod。指令格式可以自行设计。

#### 2. 实现Service抽象

Minik8s应当支持Service抽象，对pod的访问应当通过Service进行。Service需要支持至少两个pod的通信，对外提供service的虚拟ip。用户能够通过虚拟ip访问Service，由minik8s将请求具体转发至对应的pod。此外，Service内的pod也可以通过虚拟ip访问其他Service。

用户能够通过yaml配置文件来创建service，配置文件里应当指定以下内容（同理可以自行设计yaml格式，内容包括要求即可。同样强烈建议参考的kubernetes写法进行设计），包括：

- kind: 即配置类型，应该为service

## 基本功能要求

### 1. 实现Pod抽象，对容器生命周期进行管理

Minik8s需要支持pod抽象，可以通过yaml来对pod进行配置和启动。

基于pod抽象，Minik8s应该能够根据用户指令对pod的生命周期进行管理，包括控制pod的启动和终止。

pod内需要能运行多个容器，它们可以通过localhost互相访问。

用户能够通过pod的yaml配置文件指定容器的参数，包括：

- kind: 即配置类型，应该为pod
- name: pod名称
- 容器镜像名和版本
- 容器命令 (command)
- 容器资源用量 (如1cpu, 128MB内存)
- volume: 共享卷
- port: 容器暴露的端口

yaml的格式可以自行设计，包含上述内容即可，可以自行修改或添加新字段和新内容。建议参考kubernetes的写法进行设计。

Minik8s可以通过get pod, describe pod之类的指令获得pod的运行状态，展示的运行状态信息和kubernetes类似（包括pod名，运行时间、运行状态等）。可以同时运行多个pod。指令格式可

## 考核方式

该Lab自由度较高，minik8s的实现不对编程语言、实现方式等进行限制（虽然还是强烈建议参考kubernetes原本的实现方式），主要通过答辩进行考核，对项目的功能进行验证。

### 阶段性考核

该lab要求分多次迭代完成，并且会阶段性组织答辩考核，一共包括一次过程答辩和一次最终答辩。

过程答辩主要是通过助教判断一下流程进度，对流程进度进行评价。过程答辩之后，需要提交一个中期文档，汇报完成进度。

最终答辩需要完成所有功能。

评分标准：功能要求 80% + 工程要求 20%。

④

### 组织/工程要求

- 该lab为3人小组合作完成，并且指定一人为组长。自由组队
- 中期DDL暂定五月初，结题DDL暂定六月初。
- 组员内部必须明确分工。在项目开始时，开题需要指定每一次迭代的任务内容划分，人员的分工安排。中期答辩需要介绍每次迭代的完成情况，介绍实际的人员分工，以及对进度进行评估。每一个迭代结束后，按需对下一个迭代的计划进行微调。
- 按照开源社区的标准流程开发：每个功能需要通过git分支单独构建，实现完成后通过PR的方式融入主分支中。
- CI/CD：git push需要通过CI/CD中的测试。CI/CD可以任选框架（travis，Jenkins等均可）

### 开题文档要求

开题报告通过pdf格式提交，需要包括以下内容：

- 人员组成：组员的姓名、学号信息，组长指定。
- 选定的可选题目内容
- 任务的时间安排，将时间分成几次迭代，指定每次迭代需要完成的任务有哪些。
- 人员分工

## 容器虚拟化

上节课我们讲了容器可以认为是第二代虚拟化技术，其实现在容器已经比较广泛了。我们这边讲的更多是用户态容器是怎么实现的。

上节课我们讲 Cloud 的概念，云是比较大的分布式系统。

## 回顾 : Cloud Computing

A modern approach of deployment

- Cloud server instead of local server
- On-demand leasing ,
- Fast and low-cost server upgrades

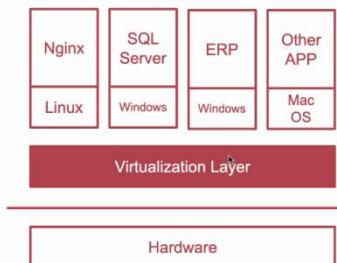


云计算是租借式的业务模型，按需获取机器、交付租金，好处是 server 的 upgrade 是比较方便的，换一台 server 即可。虚拟化是云里非常重要的支撑技术。

## 回顾 : 虚拟机为什么行，为什么不行？

### 关键技术 :

- 新的一层软件层
  - 运行在OS下方
  - 硬件上方
- (较为 ) 完整地模拟真实硬件



## 回顾 : 虚拟机为什么行，为什么不行？

### 优势

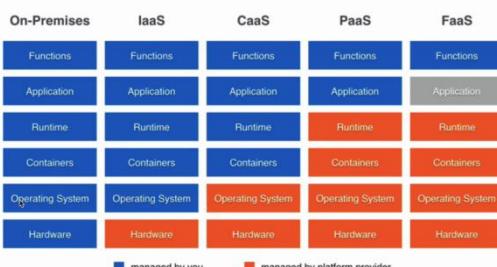
- 相比物理机方案，更好的资源利用率
- 辅助应用程序开发（大家之前的Lab）
- 简化服务器管理
- 强隔离性、结合硬件扩展实现较好的性能

### 不足

- 比较大的内存占用等资源开销（memory footprint）
- 较慢地启动时间等

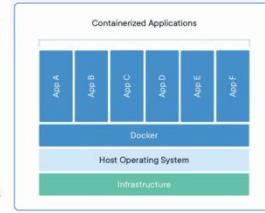
虚拟机的隔离性也比较强。虚拟机一直以来有一个问题，大家通常认为现在在物理机上加了一层抽象，会不会性能比较差了。做虚拟化的人会认为没有太大额外性能开销。因为我们模拟整个机器，包括其中的 OS，所以内存占用会比较大、启动也会慢一些。

## 虚拟机在越来越轻的云应用中显得太重



## 什么是容器？（用户视角）

- “Build , Ship, and Run”
- 应用的“打包机制”（Packaging）
- 没有虚拟化开销的VM
- 隔离环境
  - 和Host机器非常像
  - 应用直接运行在宿主机上
- 容器配置、构建、共享、部署都很方便

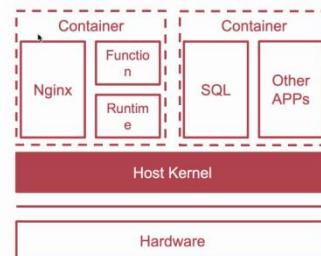


有点像“一次编译、处处执行”。可以看到 Docker 管理的东西里是没有 OS 的，OS 是在 Docker 以外的。它的隔离环境是和 HOST 机器比较像的。虚拟化里很难共享内存，容器的隔离更像比较强的进程和进程的隔离。容器给大家最直观的感受还收构建、共享、部署的方便性。

以前大家做 LAB 有部署的问题，现在有容器以后基本上大家配置上去都可以直接跑。网上有很多镜像仓库。

## 什么是容器？（系统视角）

- 容器就是一组系统的进程，是直接运行在同一个宿主机内核上方的
- 容器通过利用大量的操作系统内核特性，来实现资源隔离等目的
  - cgroup
  - namespace



对于 OS 来说，容器其实就是一个个进程。容器里的进程和平时我们用的进程有什么区别呢？容器用了很多 OS 特性来实现隔离的特性。简单来说，文件系统其实是 OS 提供了不同命名空间，每个容器运行在不同的 mount namespace 里，这样容器和容器之间看到的是不同的文件系统去用。而 cgroup 可以配置容器占据 CPU 的份额。

## 容器的好处

- 比较轻量，能够实现较快的启动时间等
  - 不需要加载和启动一个独立的客户机内核
- 接近原生执行的应用性能表现
- 较少的资源占用（如memory footprint）

容器和容器之间是共享 OS，所以启动容器的时候没有加载内核的过程。我们再讲一下容器的历史，

## 容器的历史 (1)

- 1979: Unix V7
  - (大家都还没出生呢)
  - Unix V7的开发中，引入了chroot的系统调用
  - Chroot会修改进程，及其子进程的root目录(/) → 不同的进程能够有不同的文件视角
  - BSD在1982年引入了chroot
- 2000: FreeBSD Jails
  - FreeBSD Jail允许Admin将机器划分成更小的单元，叫做Jail
  - 每个Jail能够有独立的IP地址和系统配置

## 容器的历史 (2)

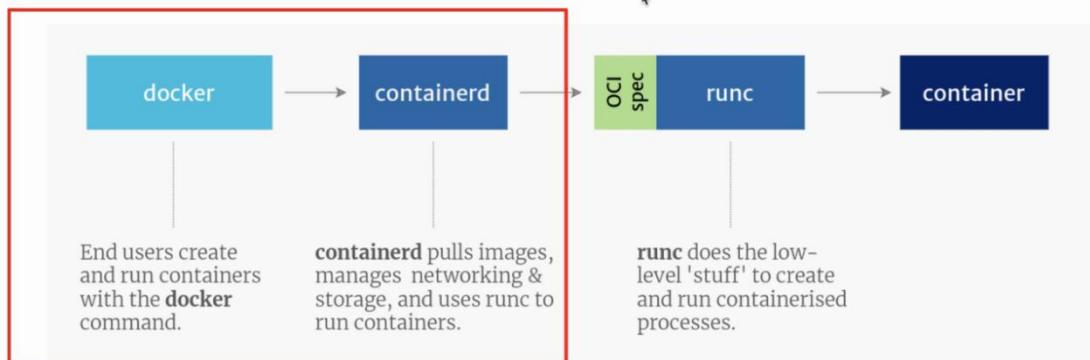
- 2004: Solaris Containers
  - Solaris Container结合了系统的system resource control以及基于zone的boundary separation的技术实现容器抽象
  - 利用了ZFS的Snapshot和Cloning
- 2006: Process Containers
  - 由谷歌在2006年发起
  - 主要目的是为了限制、统计、和隔离一组进程的资源使用 (CPU, memory, disk I/O, network)
  - 现在被改名为：控制组 Control Groups (cgroups)，Linux kernel 2.6.24.

## 容器的历史 (3)

- 2008: LXC
  - LXC (LinuX Containers)是第一个较为完整实现的Linux容器管理引擎
  - 基于cgroups 和 Linux namespaces
  - 可以直接运行在Linux系统上，不需要任何额外补丁
- 2013: Docker
  - 正是容器技术较为成熟，并且获得大量关注和需求的时候
  - 早期使用了LXC，但是在后续的开发中转向了自己的libcontainer等组件
  - 构建了十分完善成熟的容器生态

# 容器设计

- 容器的设计经过了快速的迭代
- Docker、K8s等的大规模部署是其重要推动力



## 本节课内容

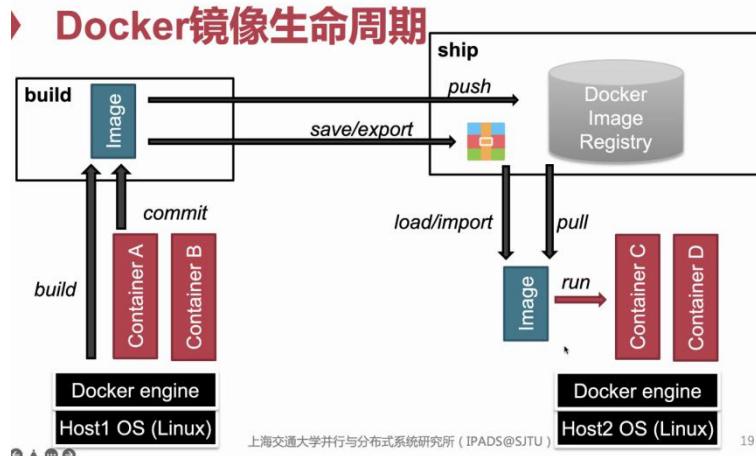
## 容器镜像的设计和实现

### 容器镜像

- Docker的“Build , Ship, and Run”是由Docker镜像 ( Docker image ) 来支撑的
  - 包含应用的运行时环境和应用
- 典型的镜像表示方法：
  - SJTU-dockerhub.com/|PADS/cloud-OS:latest
    - Repository , 类似Git仓库
    - Namespace , 类似github中的用户或组织
    - Tag , 一般用来表示不同版本

这些命令我们平时也会用。这些命令背后就是 **docker image**，包含了应用的运行时环境和应用本身。

当我们已经下载好一个镜像了以后，可以通过名字和 TAG 来确定是哪个版本的。



小明自己在一台机器上有 docker engine，他做了 build/commit 来创建了一个容器镜像。我们还有一个公共的容器镜像仓库，我们可以通过 save/export/push 到仓库里。下载下来以后，就可以通过 docker run 的方式变成新的容器。image 的本质是一个只读的模板，还是以文件系统的形式存在。

## Docker镜像

- Docker Image本质上是一个启动容器的只读模板
- 提供容器启动所需的根文件系统 ( rootfs )



我们可以把 container 展开然后去看，就会有 bin/proc/home 这种东西，和我们平时的 OS 没什么区别。我们可以通过 docker inspect 查看一下包含什么。

## Docker镜像：除了rootfs还包含什么？

要查看的镜像名，这里省略了registry和namespace

试试docker inspect 吧

老王

各种构建的记录信息

镜像的具体运行配置，如环境变量等

```
$ docker inspect busybox:latest
[{"Id": "sha256:219ee5171f8006d1462fa76c12b9b01ab672dbc8b283f186841bf2c3ca8ec93", "RepoTags": [ "busybox:latest" ], "RepoDigests": [ "busybox@sha256:bde48e1751173b709090c2539fdf12d6ba64e88ec7a4301591227c925f3c678" ], "Parent": "", "Comment": "", "Created": "2020-12-03T22:19:53.402932437Z", "Container": "6c0f2f12a00b51ce0bd0ecb9d5d6a12721e79b018ca94156862976ed070", "ContainerConfig": { "Hostname": "6c0f2f12a00b", "Domainname": "", "User": "", "AttachStdin": false, "AttachStdout": false, "AttachStderr": false, "Tty": false, "OpenStdin": false, "StdinOnce": false, "Env": [ "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin" ] }
```

包含很多构件的一些信息，更像元数据，下面还有一些配置信息。容器 docker run 的时候，需要指定一些运行的脚本等，会影响容器运行时的行为，我们主要看到的是环境变量。

## 怎么获得/创建一个镜像？

- 直接下载别人准备好的镜像：

`docker pull busybox` (使用默认的镜像仓，tag默认为latest)

- 通过压缩包保存和导入镜像：`docker save/load`

`docker save -o busybox.tar busybox`

`docker load -i busybox.tar`

- 怎么创建一个全新的镜像呢？

– Dockerfile

## Dockerfile

创建容器可以通过 dockerfile。

### Dockerfile

- 用来描述Docker镜像构建的文件

– 镜像依赖于什么样的操作系统环境？什么发行版？语言环境？

- 从本地目录中拷贝文件到镜像中

– 数据文件、可执行文件、配置文件等等

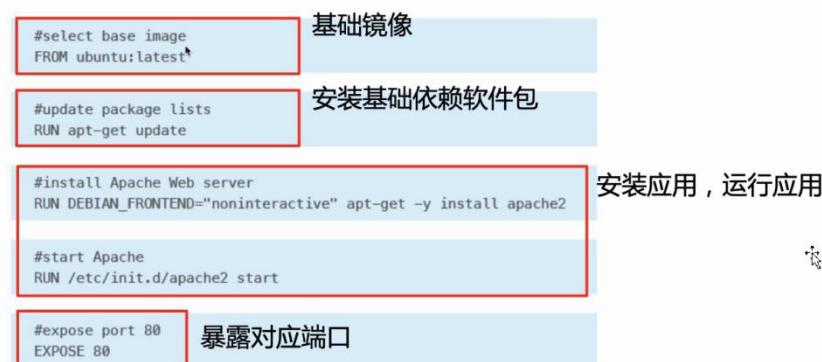
- 配置环境变量、工作目录、入口点程序

- 其他

– 指定默认的用户，端口信息，需要在环境中预先执行的命令,etc.

## Dockerfile (2)

- Apache HTTPD Server



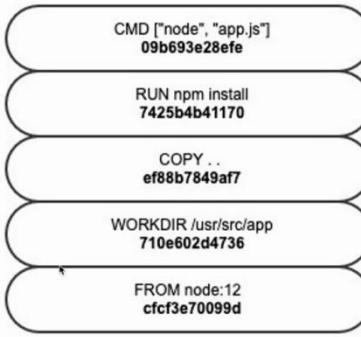
RUN 后面都是在 shell 里可以运行的指令，EXPORT 就是这个 HPPD Server 的暴露端口是 80。

## 分层镜像设计

- Dockerfile中的：*From ubuntu:latest*
  - 是怎么做到的？
- 老王给小明提了一个需求：
  - 很多的应用，比如node.js的应用，他们都依赖某一个特定版本的nodejs
  - 如果每个node.js都自己构建一个node.js环境的话，显然是低效的
  - 能不能有一个node.js的基础镜像被所有的node.js的应用镜像复用？

我们先来关注 From 是怎么做到的？我们把相同的东西放在一起合在一起，减少重复占用。

```
bash-3.2# docker build -t nodejs-helloworld .
Sending build context to Docker daemon 2.086MB
Step 1/5 : FROM node:12
12: Pulling from library/node
419e7ae5bb1e: Already exists
848839e0cd3b: Already exists
de30e8b35015: Already exists
258fdea6ea48: Already exists
ddb75eb7f1e9: Already exists
7ec8a667334: Already exists
3366ea2fc4ca: Already exists
48116fadad2c: Already exists
27e46094f3f2: Already exists
Digest: sha256:d0738468fc7cedb7d260369e0546fd7ee873:
Status: Downloaded newer image for node:12
--> cfcf3e70099d
Step 2/5 : WORKDIR /usr/src/app
--> Running in eb64a62ff878
Removing intermediate container eb64a62ff878
--> 710e602d4736
Step 3/5 : COPY .
--> ef88b7849af7
Step 4/5 : RUN npm install
--> Running in 38d9a65efc69
audited 50 packages in 2.342s
found 0 vulnerabilities
Removing intermediate container 38d9a65efc69
--> 7425b4b41170
Step 5/5 : CMD ["node", "app.js"]
--> Running in 5467db04c1a7
Removing intermediate container 5467db04c1a7
--> 09b693e28efe
Successfully built 09b693e28efe
Successfully tagged nodejs-helloworld:latest
```



最终编译出来的镜像可以看成是一系列操作，  
每个操作构成的镜像累加而成的

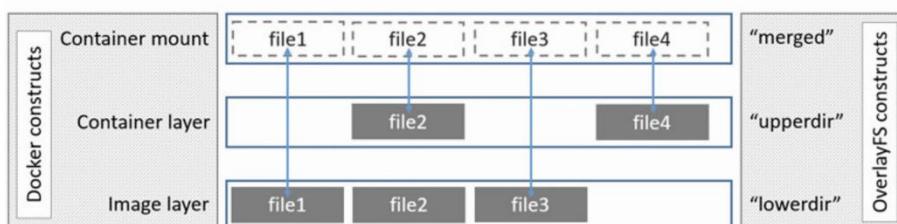
最底层的node:12可以被大量应用复用

27

library/node 可能就包括了多个版本的镜像，有些新的镜像就要下载。最终打包成一个新的镜像包。我们在这个过程中不停地创建新的镜像，最后到结束的时候才会 successfully built。这些镜像在最底层可能都是 nodejs 12，这样我们就可以分层复用它。

## › OverlayFS设计

- 运行时，怎么保持镜像文件的只读，且不影响多个容器实例对本地数据的读写操作呢？
- OverlayFS：分层文件系统，在FS上实现的CoW



每一层都是只读的，如果发生了写，就进行一个写时复制。

## 其他常用的Docker镜像操作

- 在运行的容器基础上创建镜像

*docker commit*

- 查看当前系统的镜像

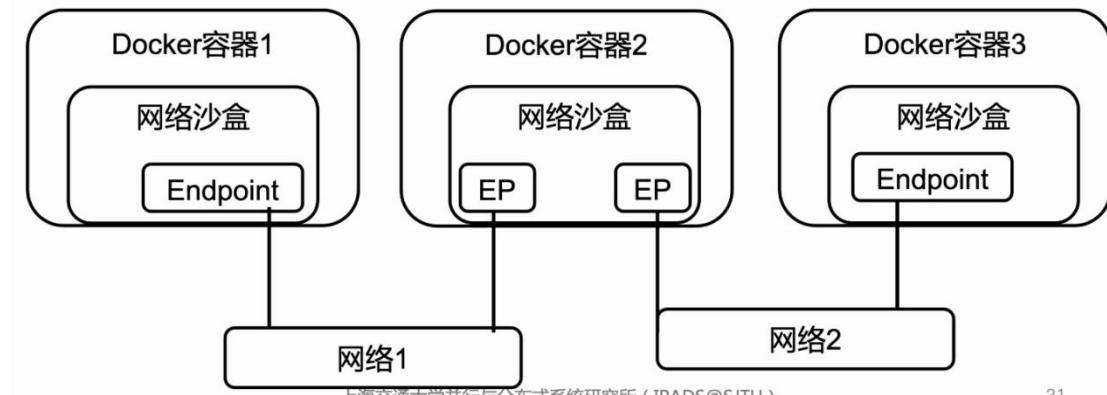
*docker images*

我们在分层文件系统上修改了这个文件，commit 以后，我们就可以使用 commit 之后的新镜像。

## 容器网络

### 容器网络：基本模型

#### CNM ( Container Network Model )



上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

31

我们先讲一些基本的网络。和虚拟机比较类似，一台物理机上可能跑 100 个容器。我们只有一个 OS，需要在机器内部配置网络让容器和容器相连。我们需要配置不同的网络让它们能连，这里比较重要的概念就是沙盒。

也就是配置一个隔离的网络环境。

## 容器网络：基本模型

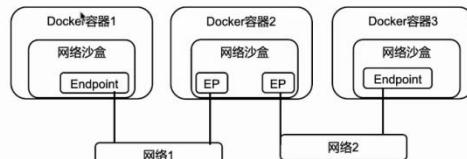
K8s中的CNI模型同样十分常用，会在后续课程介绍



老王

- **CNM ( Container Network Model )**

- 沙盒：隔离的网络运行环境，保存了容器网络栈的配置，包括了对网络接口、路由表、和DNS配置的管理（Linux上基于Network Namespace实现）
- Endpoint：用来将沙盒引入网络，通常是一对veth或者OVS内部端口
- 网络：包括一组能互相通信的endpoint，通常是Linux bridge, vlan等



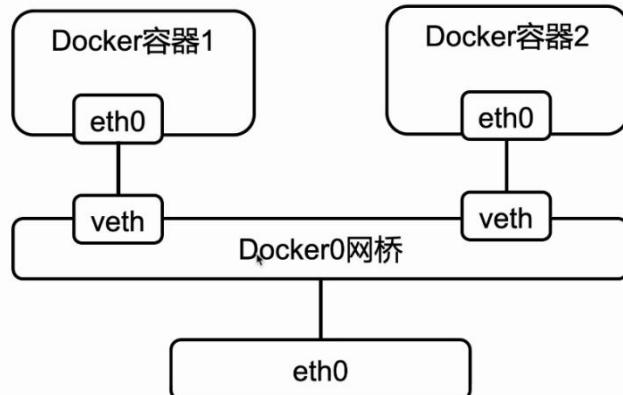
可以知道的是，大家独立配置以后，就可以有虚拟网卡这样的东西，里面就会有 DNS, 路由表等。通过 Endpoint 我们使用网络进行连接。K8s 里有更多有实际意义的需求，比如容器配置好之后就自动让容器相连。

## 容器网络：常用的网络模式

- **Bridge**：使用网桥网络配置
- **Host模式**：使用Host网络配置
- **None**: 不为容器配置任何网络功能
- **其他**

## Bridge模式：Docker下的NAT

- Docker启动时会默认一个Linux网桥Docker0
- 容器启动时，Docker会创建一对veth设备
- Veth一端挂载docker0上，另一端挂载在容器中
- 实现容器与主机通信



网桥就是有 docker 配置的网桥。

## ▶ Bridge模式：Docker下的NAT

- Docker通过iptables配置底层网络规则

```
$ sudo iptables -vnL -t nat
Chain PREROUTING (policy ACCEPT 178K packets, 32M bytes)
pkts bytes target     prot opt in     out      source         destination
  37 2220 DOCKER     all  --  *       *       0.0.0.0/0      0.0.0.0/0      ADDRTYPE mat
ch dst-type LOCAL

Chain INPUT (policy ACCEPT 178K packets, 32M bytes)
pkts bytes target     prot opt in     out      source         destination

Chain OUTPUT (policy ACCEPT 9932 packets, 754K bytes)
pkts bytes target     prot opt in     out      source         destination
  0    0 DOCKER      all  --  *       *       0.0.0.0/0      !127.0.0.0/8   ADDRTYPE mat
ch dst-type LOCAL

Chain POSTROUTING (policy ACCEPT 9932 packets, 754K bytes)
pkts bytes target     prot opt in     out      source         destination
  0    0 MASQUERADE  all  --  *       !docker0  172.17.0.0/16  0.0.0.0/0
  0    0 MASQUERADE  all  --  *       !br-3f56f4f507ae 172.18.0.0/16  0.0.0.0/0
  0    0 MASQUERADE  all  --  *       !br-2fe75e29dad8 172.19.0.0/16  0.0.0.0/0

Chain DOCKER (2 references)
pkts bytes target     prot opt in     out      source         destination
  0    0 RETURN      all  --  docker0 *       0.0.0.0/0      0.0.0.0/0
  0    0 RETURN      all  --  br-3f56f4f507ae *       0.0.0.0/0      0.0.0.0/0
  0    0 RETURN      all  --  br-2fe75e29dad8 *       0.0.0.0/0      0.0.0.0/0
```

给大家留一个小实验，在课后查看一下安装了 docker 的环境下的iptables



老王

如果我们在 docker 环境下有 IPTable 会怎么样。

## 容器网络：常用的网络模式（2）

- Host: 与主机共享root network namespace**
  - 容器有完整的权限可以操纵主机协议栈、路由表、防火墙等
  - 启动时 `-net=host`
  - 容器权限特别高，要非常谨慎使用（最好不用！）

这和虚拟机类似，在 host 共享的情况下，容器的权限会更强。平时还是不要用这样的方法。

# 容器网络：常

- Host: 与主机共享

- 容器有完整的权限

- 启动时 `--net=host`

```
$ docker run --rm -it --net=host busybox
/ # ifconfig
br-2fe75e29dad8 Link encap:Ethernet HWaddr 02:42:33:1A:5D:44
    inet addr:172.19.0.1 Bcast:172.19.255.255 Mask:255.255.0.0
        inet6 addr: fe80::1/64 Scope:Link
        inet6 addr: fc00:f8f3:cc:793::1/64 Scope:Global
        UP BROADCAST MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

br-3f56f4f507ae Link encap:Ethernet HWaddr 02:42:92:80:08:00
    inet addr:172.18.0.1 Bcast:172.18.255.255 Mask:255.255.0.0
        UP BROADCAST MULTICAST MTU:1500 Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet HWaddr 02:42:17:0.2 Bcast:172.17.0.1
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:16 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2436 (2.3 KiB) TX bytes:0 (0.0 B)

lo       Link encap:Local Loopback
          inet addr:127.0.0.1 Mask:255.0.0.1
          UP LOOPBACK RUNNING MTU:65536 Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

/ #
```

正常启动

37

加了`--rm`之后，容器就会自己消失。

## VOLUME 与容器持久化

## 容器引擎的新需求：怎么持久化数据？

通过现在的容器镜像和网络，  
我已经可以在容器中计算任  
务了。  
但是每次退出容器数据就丢  
失了。



老王



怎么持久化数据呢？

小明

退出以后，哪怕容器没有了，但是数据是持久化下来的。

## 容器卷 : Volume

- 为了持久化保存和共享数据，Docker提出了卷 ( Volume ) 的概念
- 卷是文件或目录，能够被挂载到容器内部，在容器被删除后仍然能够访问
- 两种主要方式
  - 数据卷
  - 数据卷容器

其实还是文件系统这一套东西，是通过挂载的方式挂载进容器里，容器删除以后，我们通过 host 还是可以访问。

## 容器卷 : 数据卷

- 可以在容器之间共享和重用
- 对数据卷的修改会立马生效
- 对数据卷的更新，不会影响镜像
- 卷会一直存在，直到没有容器使用

## 容器卷 : 数据卷

- 怎么创建一个数据卷？
  - Docker run的时候，可以通过-v在容器内部创建一个数据卷

```
docker run -dp --name ipads-web -v /ipads-web ubuntu:20.04
```
  - 挂载一个主机上的目录到容器作为数据卷（主机中的~/ipads-web）

```
docker run -dp --name ipads-web -v ~/ipads-web:/opt/ipads-web ubuntu:20.04
```

    - 从主机挂载单个目录作为数据卷

```
docker run -dp --name ipads-web -v ~/.bash_history:/.bash_history ubuntu:20.04
```

每次我们主机里有一个固定的目录挂载到容器里进行使用，这样就可以在 docker run 这一步去做。

## 容器卷 : 数据卷容器

- 场景：用户需要在容器之间共享一些持续更新的数据
- 数据卷容器：一个普通的容器，专门用来提供数据卷供其它容器挂载

1. 创建一个数据卷容器，叫做vcont，它包含一个/vcont的数据卷  
`docker run -it -v /vcont --name vcont ubuntu:20.04`

2. 在其他容器中使用--volumes-from来挂载vcont容器中的数据卷：  
`docker run -it --volumes-from vcont --name test1 ubuntu:20.04`  
`docker run -it --volumes-from vcont --name test2 ubuntu:20.04`

# 容器安全

## 容器安全

- Linux Capability
- Seccomp
- ulimit
- 其他机制：SELinux、AppArmor等

容器安全这一块简略地讲一讲。在我们 OS 里也会讲一些 capability 的内容。容器相比进程肯定是更安全的，它基于很多现有的安全机制对安全进行加固的。

### 容器安全: Linux Capability

- Linux这样的内核中，通常存在用户的概念
  - 最常见的是普通用户和root用户
  - 通常root用户会拥有所有的权限



Linux Capability !

小明

Root权限太大了，普通用  
户权限太小了。  
为了让普通用户权限执行某  
些操作只能提前到root？



老王

capability 其实就是把普通用户和 root 用户进行更细粒度的分类，比如 linux sudo 可以变成 root 用户去做一些事情。但是问题就是 root 的权限太大，我们希望在中间找一个中间权限，这就是 Linux Capability。

### 容器安全: Linux Capability

- Linux 2.2之后引入的内核特性
- 将特权用户（比如root）的权限进行细粒度的拆分，变成不同的capability
- 可以通过给进程某些特定的capability使其完成对应的特权用户才能完成的操作

这个概念没有一个特别好的具体定义，capability 可以认为把资源进行拆分通过 token 进行管理的方法。fd 也可以认为是对一个文件的某些权限，比如以读写方式打开的文件就可以认为 fd 是一种 token。

## 容器安全: Linux Capability

- Linux 2.2之后引入的内核特性
- 将特权用户（比如root）的权限进行细粒度的拆分，变成不同的capability
- 可以通过给进程某些特定的capability使其完成对应的特权用户才能完成的操作

### CAP\_CHOWN

Make arbitrary changes to file UIDs and GIDs (see [chown\(2\)](#)).

<https://man7.org/linux/man-pages/man7/capabilities.7.html>

## 容器安全: Linux Capability

- Docker中，可以通过--cap-add和--cap-drop来为容器增加或减少capability

### 案例：默认有CAP\_CHOWN

```
$ docker run --rm -it busybox
/ # chown 2:2 /etc/hosts
/ #
```

当drop CAP\_CHOWN之后，chown操作会失败

```
$ docker run --rm -it --cap-drop=chown busybox
/ # chown 2:2 /etc/hosts
chown: /etc/hosts: Operation not permitted
```

这就是比较细粒度的权限了，比如我们限制 chown 命令不能使用。一般来说，平时我们使用 docker 不太会使用到 capability。

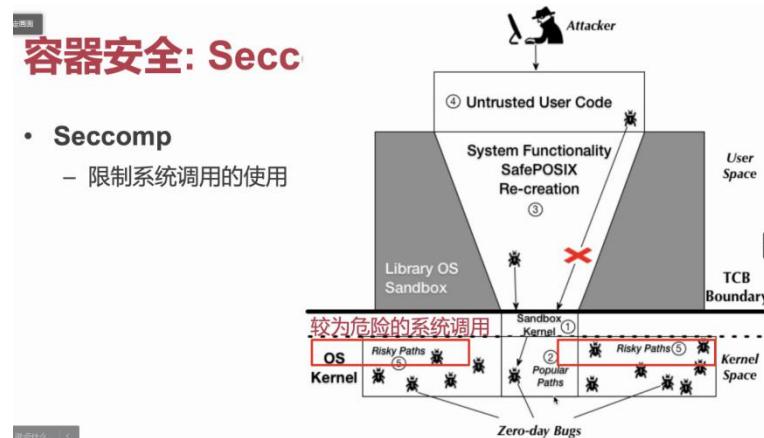
## 容器安全: Linux Capability

- Docker等容器引擎维护了默认的Capability的清单
  - 核心原则是：最小权限
- 
- 通常不建议给一个容器--privileged，而是应该针对具体的需求，分配细粒度的Capability

这种情况出现在，我们对容器内运行的应用有怀疑的情况下。

## 容器安全: Seccomp

- 除了Capability，也就是容器能够执行的操作
- 另一个明显的容器需要面对的攻击面是：**内核系统调用**
- 最新的Linux kernel已经支持400左右的系统调用了，其中一些系统调用很有可能触发内核的bug



## 容器安全: Seccomp

- Seccomp**
  - 限制系统调用的使用
  - 内核底层使用BPF/eBPF技术实现
- Docker**
  - 在启动容器的时候，会根据一个默认的系统调用的黑白名单，来配置seccomp，限制执行危险的系统调用

## 其他容器引擎

- 除了Docker之外，当前还有大量的其他容器引擎可供使用
- openEuler开源的Isulad

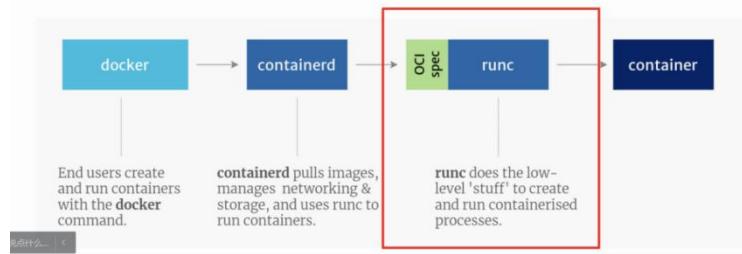


- podman



## 下节课：容器运行时与轻量级虚拟化

- 容器运行时：容器引擎的引擎
- 怎么进一步增强容器安全？轻量级VM虚拟化



2022/4/1

上节课我们介绍了容器内部的一些实现，比如镜像怎么设计实现（Build-Ship-Run）。

## 回顾（2）：Docker与容器引擎

- 卷与持久化数据
  - 什么是数据卷和数据卷容器？
  - 他们的异同是什么？
- 容器安全
  - Capability机制
  - Seccomp机制
- OCI与RUNC

最后讲了一些安全，讲了 Capability 机制和 Seccomp 机制。

### 容器安全：Linux Capability

- Linux 2.2之后引入的内核特性
- 将特权用户（比如root）的权限进行细粒度的拆分，变成不同的capability
- 可以通过给进程某些特定的capability使其完成对应的特权用户才能完成的操作

**CAP\_CHOWN**  
Make arbitrary changes to file UIDs and GIDs (see `chown(2)`).

<https://man7.org/linux/man-pages/man7/capabilities.7.html>

`CAP_CHOWN`，这个 capability 对应了 `change owner` 这个命令。`copy` 过来一个文件修改成自己的以后，我们就可以对文件更方便地进行操作了。但是如果我们可以任意地修改文件的话，之前的 `owner` 的保护就没有意义了，所以我们可以对进程设置 capability 的选项。

## 容器安全: Linux Capability

- Docker中，可以通过--cap-add和--cap-drop来为容器增加或减少capability

案例：默认有CAP\_CHOWN

```
$ docker run --rm -it busybox  
/ # chown 2:2 /etc/hosts  
/ #
```

当drop CAP\_CHOWN之后，chown操作会失败

```
$ docker run --rm -it --cap-drop=chown busybox  
/ # chown 2:2 /etc/hosts  
chown: /etc/hosts: Operation not permitted  
/ #
```

上节课没有系统的讲容器是什么东西。

## 小明的问题：容器是什么？

- Docker：
  - 底层基于Linux命名空间和控制组提供隔离
  - 容器镜像 ( Dockerfile、仓库等 )
  - 容器网络等



小明

其实 docker 可以认为是基于容器镜像，底层使用了 cgroup 和 namespace 提供隔离，当然还有一些容器网络之类的东西。我们可以这样想，这些功能没有限制它是怎么实现的。

如果有一个平时常用的虚拟机：

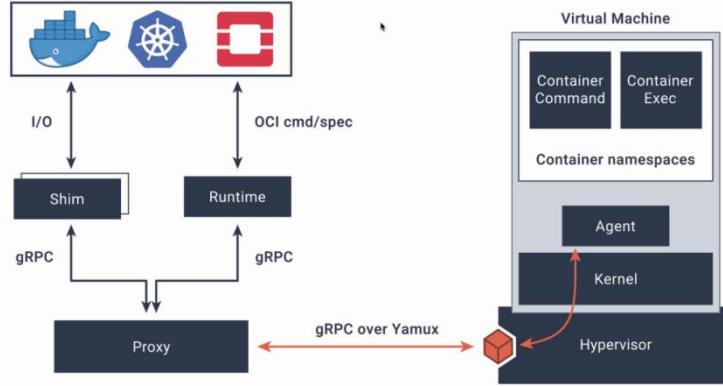
如果一个虚拟机：

- 能够直接跑容器镜像
- 能够提供Docker支持的API
- 能够保证和Docker类似甚至更强的隔离
- 能够有容器的网络连接

所以容器并没有绑定在 docker 上，真有人这么做了。

## KataContainer

### KataContainer : 基于虚拟机的容器实现



docker 中没有引入虚拟机的概念，没有做成一个完整的虚拟机。而 KataContainer 中是有虚拟机的，我们可以看到是有一个 Hypervisor 的，所有东西都跑在虚拟机里，当然我们外界可以有 proxy 和 docker 等交互。

### 小明的问题：容器是什么？

- 随着各种不同隔离技术的发展和对容器生态的兼容，容器不仅仅是Docker和基于Linux Container的系统
- Docker
  - 底层基于Linux命名空间和控制组提供隔离
  - 容器镜像 ( Dockerfile、仓库等 )
  - 容器网络等
- KataContainer
  - 底层基于虚拟化技术提供隔离
  - 容器镜像 ( Dockerfile、仓库等 )
  - 容器网络等

共性部分

上节课我们也提到了，虽然 docker 很强，但是不是只有它们一家，并且相比 docker，各有各的优势，比如 KataContainer 因为是基于虚拟机的，它能提供的隔离性更强。所以在公有云的环境中，可能 KataContainer 更受欢迎。

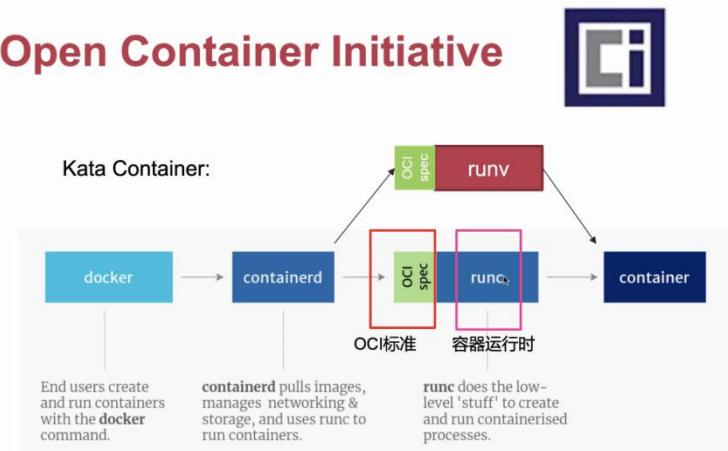
### Open Container Initiative



- Linux基金会下的子机构
- 和具体的厂商之间独立
- 目前主要负责维护：
  - 容器运行时标准：将具体的隔离方式从容器中解耦出来
  - 容器运行时参考实现 ( reference runtime )
  - 容器镜像格式标准

那么提取这些共性部分呢，很有可能这些东西就成为了一种大家都使用的标准。比如镜

像的相同标准可以使得 KataContainer 和 Docker 的镜像兼容。这个子机构是 Linux 下的，主要维护了 Docker 的运行时标准、镜像格式标准，这是容器可移植的关键。比如我们学 OS 的时候有 EOF 格式，容器镜像也一样有通用的格式，这样我们就可以在不同的实现中做迁移。



`docker` 后是 `containerd` 管理这些容器和镜像，是后台的引擎。再往后就是和真正容器相连的，OCI 标准。OCI 对应的就是运行时的一些标准和镜像的一些标准，有了这些标准以后，镜像就可以直接和运行时对接了。最后的容器运行时就是对应了一些 Linux 底层的创建命名空间和控制组的概念。

这就是接口和实现的分离，我们使用 OCI 标准兼容前台发来的请求。

`runv` 其实就是 KataContainer 中，接到对应的和虚拟机相关的事情来构建这个容器。后台还是容器的概念，但是实际的实现是差距比较大的。所以 OCI 带来的好处就是可以兼容 `docker` 的指令。

大家做了 miniK8s 以外，还可以做容器的一些探索。这里可以给大家一个小作业，比如我们把后台换成不同的运行时怎么做。

## OCI Spec

### OCI Spec: 指定了容器声明周期管理接口

- 创建、运行、通知、删除、查询状态

OCI interfaces	Description
<code>state &lt;sandbox-id&gt;</code>	Query the state of a sandbox.
<code>* create &lt;sandbox-id&gt; &lt;func-id&gt;</code>	Create sandbox with the ID and bundle path, a config.json file is required to indicate details.
<code>start &lt;sandbox-id&gt;</code>	Run a created sandbox.
<code>kill &lt;sandbox-id&gt; &lt;signal&gt;</code>	Send a signal to a created/running sandbox.
<code>delete &lt;sandbox-id&gt;</code>	Delete a sandbox.

容器运行时接口简化（来自Molecule , ASPLOS'22）

OCI Spec 是底层的一些接口，外面是 `docker run` 等命令。而这里是沙盒的一些概念。接口也比较直接，`state`（查看状态）、`create`（创建沙盒）、`start`（启动一个）。这都可以对应到 `docker ls`、`docker ps`、`docker kill`、`docker remove` 这些前端的接口。

# Runc

## Runc是什么？

- 基于Linux cgroup/namespace等实现的OCI容器运行时
- 作为样例实现，是现在Docker、K8s等的默认容器运行时



上海交通大学并行与分布式系统研究所 (IPADS@SJTU)

14

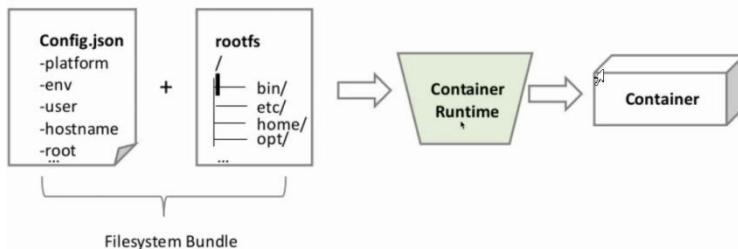
Runc 是基于控制组和命名空间等实现的 OCI 容器运行时。

## Runc怎么管理容器？

- Runc的容器依赖于两个部件
  - Rootfs
  - Config.json
- 其中，Rootfs就是Docker中镜像维护的rootfs，需要上层容器引擎提前为runc准备好rootfs目录
- Config.json是一个配置文件，包含容器运行所需的各种配置

Runc 主要依赖于 Rootfs（根目录）和一个配置文件。核心我们还是搞清楚 Rootfs。

## Runc怎么管理容器？



docker create 提供的可选配置不够，可能我们就需要深入到 runc。

## Runc : 使用案例

### 1. 创建一个目录

```
cd ${HOME}/myalpine2
```

### 2. 创建rootfs目录

```
mkdir rootfs
```

### 3. 从Docker现有镜像中生成rootfs

```
docker export $(docker create alpine) | tar -C rootfs -xvf -
```

### 4. 生成config.json spec

```
runc spec
```

在这个基础上，我们从现有镜像生成 rootfs，这部分是唯一使用 docker 指令的。alpine 是 linux 里一个比较轻量化的分支，随着容器发展而发展起来。我们创建完 alpine 的根目录并且解压出来，然后我们生成配置文件 config.json。这样我们就准备好了 rootfs 和 config.json。

## Runc : 使用案例 ( 2 )

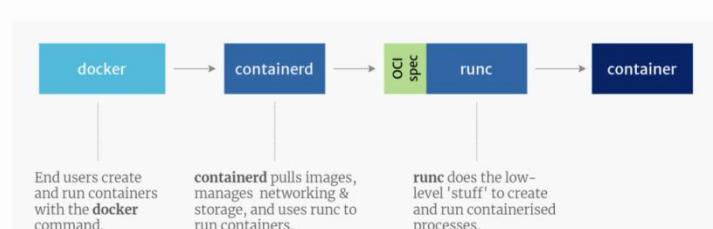
- 准备工作完成，准备运行
- 进入OCI Bundle目录（包含rootfs和config.json的目录）  
`cd ${HOME}/$myalpine`
- 运行alpine bundle  
`sudo runc run myalpine`



接下来我们进入到目录以后，运行 runc。这是一个标准的根目录，有 root, lib, home 等。

## 从Docker到Runc

- 从Docker 1.11开始，Docker引擎开始使用containerd与runc来运行容器



上节课我们也讲过 docker 的历史，docker 还是借助 linux container 的东风。后来它自己做 containerd 和 runc 跑自己的容器。

## 从Docker到Runc (2)



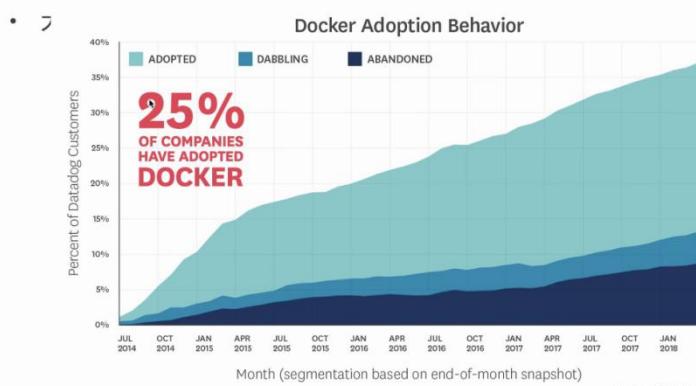
上面指令是映射一个 host 的数据卷。在 docker run 的时候，docker 会去准备这么一个 config 文件，后台会根据我们的 spec 生成我们的 config.json 文件。docker 后端再根据后端来跑这么一个 config 文件。

这里讲完了容器，简单做一下小结。

容器自己的优势有两点：

- 可移植性（Portability）好，这是相对于没有虚拟化来说的，比物理机裸机的可移植性，容器还是好很多的。容器的出现提高了应用可移植性，降低了部署的难度。
- 性能好，这是相对于虚拟机来说的。大多数时候容器的执行性能和我们在进程里没有什么差别。

## 容器小结 (2)



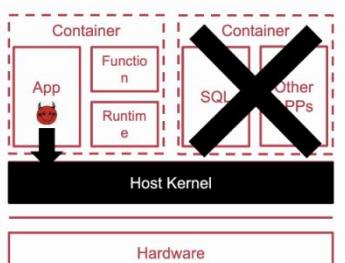
18年的时候有 25% 公司用 docker。20 年已经有 20% 的公司已经开始用 serverless 了，这基本上都是要用到 docker 的。

容器比较大的问题就是隔离（Containing），容器的问题主要就出在隔离上，但是性能和隔离有 tradeoff。

## 反思：容器有什么缺陷呢？

### • 缺陷

- 安全隔离性较弱，相信宿主机内核
- 主要是资源隔离



### • 有没有什么办法：

- 既能够实现高性能、细粒度
- 还能保障高安全性？

`container` 在针对内核的隔离方面和一个普通的进程没有什么区别，都需要相信 `host kernel`。当然容器里面和容器外面的隔离性的差别，主要还是在于资源隔离。如果我们的 APP 是坏的，去攻击宿主机内核，攻陷了以后可以把自己提权成为超级进程，那么它就可以把我们的 `container` 搞坏。

限制系统调用和 `capability` 都是在没有攻击成功的基础上，攻击成功以后这些方法就再也没用了。近些年希望结合容器的高性能和虚拟机的高安全性结合在一起。产生的就是轻量级 VM 虚拟化。

第一节课的时候，我们也介绍过一些轻量级虚拟机的实现。它们的核心都是类似的，让虚拟机往容器靠拢或者让容器向虚拟机靠拢，从而兼顾性能和安全。

## 轻量级VM虚拟化

- 什么是轻量级VM虚拟化
- gVisor系统设计与实现
- FireCracker设计与实现

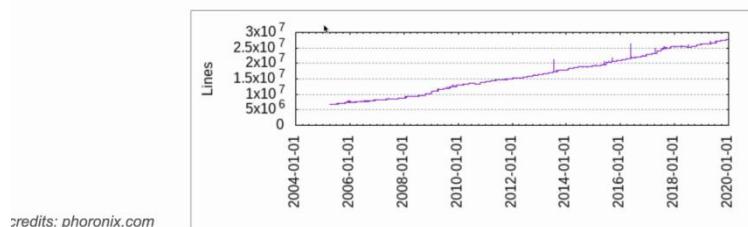
还没有达到第三代虚拟化技术的高度。

## 轻量级 VM 虚拟化

首先先从 linux 内核安全性说起，

### 单一的Linux内核有多不安全？

- 通常来说，一个软件越复杂，越大，其存在的Bug越多  
( 和代码行数成一定的比例 )
- 到2020年，Linux的代码行数到达：27.8 million lines

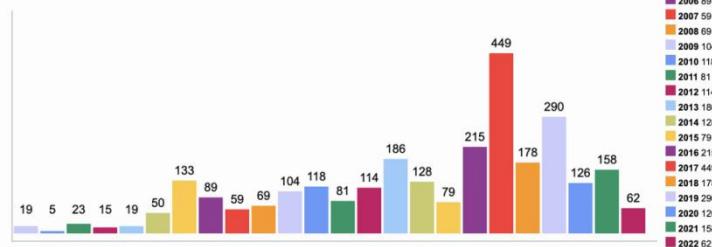


我们有一个概念叫做 TCB。Trust-computing-base：计算可信基，这是我们相信代码的量级。一般应用是相信操作系统的，也就是 Linux。Linux 内核通常是我们写代码的时候相信的部分，但是一个软件越复杂越大，拥有的 BUG 越多和代码行数成正比。一般是平均 XXX 行出现一个 bug，我们可以计算出 BUG 的期望值。

现在 Linux 越来越大，一方面是要向更早的版本兼容的问题，其次支持的硬件越来越多，我们要写各种各样的驱动。2020 年是 2700 万行。

## 单一的Linux内核有多不安全？（2）

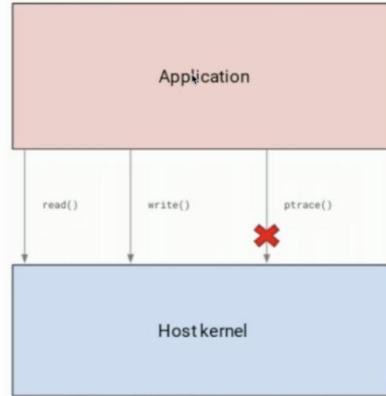
- Linux CVE：到22年，已经有2769个正式CVE
- 还有大量的未披露的Bug



有的时候硬件问题会引发恐慌，Linux 内核爆出的 bug 很多。

## SECCOMP：为什么不够？

- 有限的系统调用面
  - Seccomp的策略越严格，对于应用和系统的保护越强
- 低开销（内核直接支持）



严格的 SECCOMP，比如我们禁止应用的读写和网络的访问，比如我们喂入一些数据，这样这个应用能做的事情就非常有限。这样我们的系统调用面就会对应用产生影响。比如一个大数据处理的应用要不停地从硬盘中读数据，这样做优化保护能力是强了，但是应用场景就受限了。还有一些挑战是系统认为安全的漏洞不一定安全。

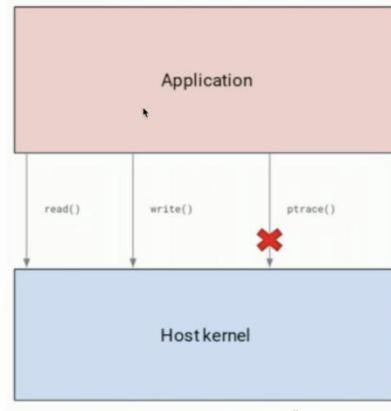
## CVE-2014-3153：安全的调用不安全

- CVE-2014-3153是一个经典的Linux内核提权漏洞，影响范围相当广泛
- Linux Futex ( Fast Userspace muTExes )
  - 加速libc层次的互斥访问，大多数情况下futex可以在用户空间就处理互斥访问
  - Linux从2.5.7开始支持Futex
  - 被大量的应用和标准库使用
- CVE-2014-3153：Futex的实现存在Bug（类似Use-after-free），允许用户通过其进行提权

以前大家普遍认为信号量 PV 和 OS 没什么关系。比如我们调用 `wait` 的时候，就陷入到 OS 里面了，OS 提供一个支持让进程等在某一个东西上面，然后再提供 `notify` 支持。应用和标准库会使用很多 Futex。

# SECCOMP：为什么不够？

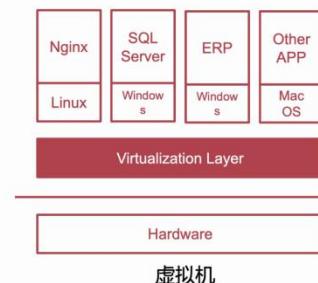
- **有限的系统调用面**
  - Seccomp的策略越严格，对于应用和系统的保护越强
- **低开销（内核直接支持）**
- **挑战：**如果应用想执行某些不被允许的调用怎么办？
- **挑战：**一些用户认为安全的系统的调用并不一定安全



如果不让应用使用锁，那么应用的多线程可能多做不了。所以我们单单封锁系统调用是不够的，但是封太多应用就受限了。

## 基于虚拟机的容器方案

- **应用程序和客户机内核/虚拟化的硬件交互**
- **不共享内核**
- **挑战：客户机和宿主机内核中存在大量的重复机制**
  - 调度器
  - 内存管理
  - **导致低效的资源管理**



我们可以认为这是容器上加固的方法，以前我们直接走容器，现在我们在每个容器里加一个OS，这样大家的内核就不共享了，攻击Linux容器里的OS就不会影响其他跑Windows的容器。但是存在很多的重复机制，比较有意思的是double scheduling。比如我们虚拟化layer的调度粒度是一个个虚拟机，它是看不到虚拟机内部的进程的。但是真正要把硬件资源交付给虚拟机里的进程，虚拟化层不知道虚拟机里的决策就会出现管理上的问题。

## 两条道路

- **目标：容器的性能、易用，但有更强的隔离性**



容器的隔离性差一些，但是虚拟化性能好。我们的目标就是结合两者的优势，让两者往中间靠。

## Google gVisor

### gVisor的目标

- **额外的一层保护来避免单一内核导致的漏洞或攻击**
  - 类似VM客户机内核
- **对应用没有限制**
  - 不存在应用A能够跑，而应用B不能跑的情况
- **使用宿主机内核的调度器和内存管理**
  - 从单一内核来管理资源，提升资源利用率等

gVisor 的核心就是在容器的基础上加一层保护。需要注意的是其实它们还是共享内核。第二个目标是对应用没有限制，这是应用可移植性的问题。最后一个是宿主机内核的调度器和内存管理。

如果之前公有云没有开起来的话，大家可以私聊助教。

### gVisor的目标

- **额外的一层保护来避免单一内核导致的漏洞或攻击**
  - 类似VM客户机内核
- **对应用没有限制**
  - 不存在应用A能够跑，而应用B不能跑的情况
- **使用宿主机内核的调度器和内存管理**
  - 从单一内核来管理资源，提升资源利用率等

怎么平衡这两点？

第一个和第三个有点矛盾的，第一个是隔离（减少共享）而第三个是增加共享，我们希望使用到 HOST 的调度器、内存和网络等。

### 隔离System API，保留剩下的

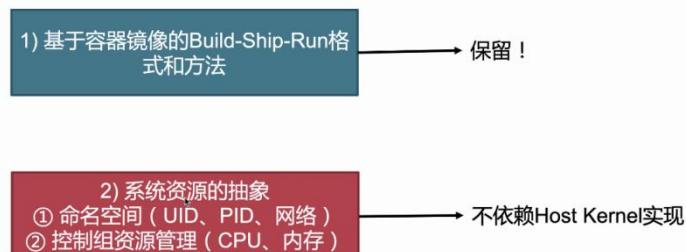
1) 管理、调度、分发物理资源（如内存、时间片）的机制 → 依赖Host Kernel实现

2) OS提供的其他抽象和System API → 不依赖Host Kernel实现

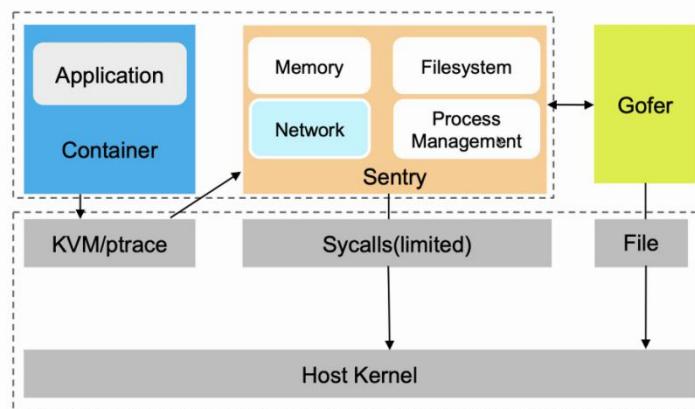
这里的核思路就是我们要知道哪些是依赖于宿主机的，哪些是我们自己可以实现（不依赖于宿主机）的。这和我们 OS 中的微内核有点相似，比如调度这个东西通常是由内核完全控制的，FS 其实可以往用户态放，这就是微内核的架构。

这里也有类似的概念，OS 的其他抽象和系统的 API 不完全依赖于 Host Kernel 去实现。比如我们的文件系统不依赖于 Host 宿主机来管理，这些功能可能就不用做隔离。

## 隔离System API，保留剩下的（容器视角）



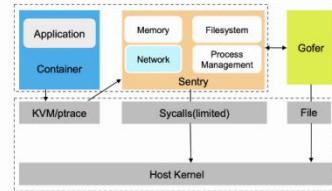
## gVisor架构



gVisor 的架构如上图。容器本身和 docker 没有太大差别。Sentry 是 gVisor 比较关键的概念，它相当于是用户态的内核。比如 ESO Kernel 很小，应用和 OS lib 链接到一起来实现一些功能。

## gVisor架构

- 应用程序的系统调用被Sentry直接处理
- Sentry: 一个用户态的内核，基于Golang语言实现
- Sentry本身会和应用程序之间隔离开 ( Ptrace , KVM等)
- Gofer: 处理文件等I/O的组件



Sentry 类似于 OS lib，里面有内存管理、文件系统（mount namespace）、网络、进程管理（pid）。还有一个关键是 Gofer 是处理文件相关的组件，Sentry 和用户是同一层的，权限不是很够。Sentry 没有权限去操作文件和网络，所以我们需要有 Gofer 的概念。现在的是我们单独实现 Gofer 模块来实现文件相关的读写。

在这个结构下，我们通过强大的用户态内核把大部分系统调用给做掉了。当然这部分还有一层关键是 libOS 很多时候是链接进来作为一个库来使用的，没有隔离的需求。而 Sentry 会有隔离的考虑，它可能是一个单独的进程，我们可以通过 Ptrace 和 KVM 来做隔离。

## Sentry怎么保证自己能够始终实现Linux接口？



Linux本身在快速发展，Sentry怎么保证自己能够始终兼容？

小明

Q: Sentry 怎么保证快速兼容 Linux 的新版本呢？Sentry 是用户态的 OS，它是架在 host kernel 上的，十几年来代码增长了五倍，兼容性是不是很难做呢？

A: 实际上并不是，系统调用 API 会很稳定。

## Sentry怎么保证自己能够始终实现Linux接口？

- 观察：Linux的系统调用API是十分稳定的

*Breaking user programs simply isn't acceptable. (...) We know that people use old binaries for years and years, and that making a new release doesn't mean that you can just throw that out. You can trust us.*  
-- Linus<sup>[1]</sup>

- 1个API可能会“永久”存在：uname, olduname, oldolduname?
- 大家会看到大量的变种：poll、epoll、dup、dup2、dup3、...

在兼容性要求下，我们很难去除一个系统调用。在这个基础上，我们做 Sentry 的兼容就没有这么难了。

## Sentry的隔离保障

- 应用不直接和Host Kernel交互

- 信息泄露的可能性会降低

- 触发Kernel的CVE的可能性会降低

- 多了一层隔离层，defence in depth
- 但是依然有可能触发

刚才我们讲过 libOS 不需要隔离，而这里应用不直接和 Host OS 交互，而是和 Sentry 交互。这里有一个防御纵深（defence in depth）的概念，也就是攻破了第一层还有第二层。通过加一层 Sentry，应用和 Sentry 交互，它本身的代码量比较小。很多功能不提供，这样 Sentry 出现 bug 的概率小于 Linux 出 Bug 的概率。这样触发 kernel CVE 的概率会降低，并且调用 host kernel 的 syscall 的频率降低了，所以触发概率也是降低的。但是依然有可能触发 kernel CVE。

## Sentry的隔离保障 (2)

### Sentry的隔离保障 (3)

- 应用不直接和Host Kernel交互

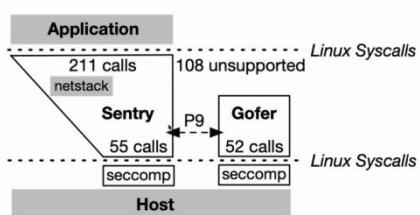
- 信息泄露的可能性会降低

- 触发Kernel的CVE的可能性会降低

- 多了一层隔离层，defence by depth
- 但是依然有可能触发

- 挑战

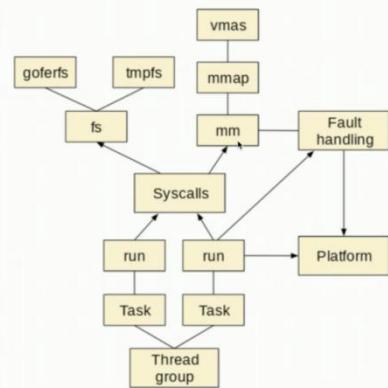
- 需要实现所有要支持的系统能力（比如大量的系统调用）
- 拦截和中转了大量的操作，性能损失



我们要实现网络，是相当复杂的。第二，我们需要拦截操作，这样每次都要经过一次 Sentry。每次的 IO 都是丢给 Gofer 去做的，所有的 IO 都必须通过 Sentry->Gofer->IO，所以它对 IO 的支持是比较差的。

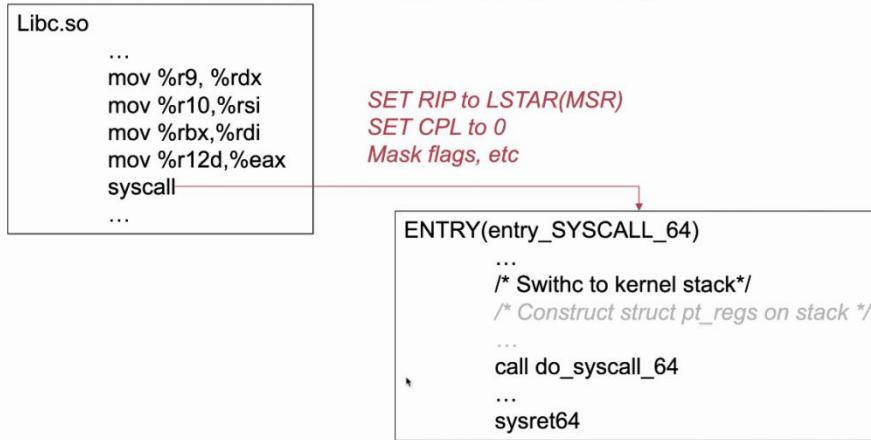
## Sentry架构

- 较为完整地实现了系统接口



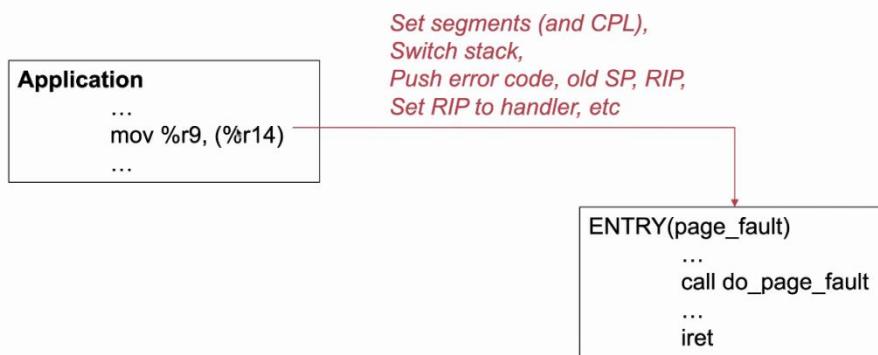
文件系统有 gofer 的文件系统，mm 提供了 mmap 等去管理 va。可以看到的是，对于内存和文件的支持是比较好的。我们刚才讨论了 Sentry 怎么为应用提供保障。那么应用和 Sentry 之间怎么做交互呢？怎么做相对比较长的隔离并且可以进行必要的交互呢？

## ▶ Linux中的Syscall处理 ( x64 )



都是有一个 entry 掉入，然后做一些事情然后返回。

## ▶ Linux中的Trap/Faults处理 ( x64 )



# Sentry与应用的交互与隔离

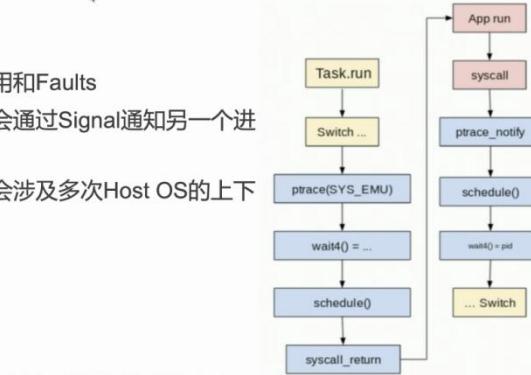
- 支持多种平台 ( Platform )
  - Ptrace
  - KVM
- 要求1：需要能够拦截syscall指令
  - 用来实现Linux中的syscall处理
- 要求2：需要能够拦截异常等
  - 用来实现Linux中的trap、fault的处理

其实 Sentry 提供了 Ptrace 模式和 KVM 模式。什么时候应用需要 Sentry 呢？也就是系统调用的时候，但是麻烦的问题是应用和 Sentry 是同级的。也就是我们一个应用要检测另一个应用的系统调用。

比如我们可以通过 `strace a.out`，它会帮我们监控 a.out 的系统调用。总的来说，我们在用户态可以通过一些 OS 支持的方法来得到这个信息。Sentry 作为一个用户态的内核进程，它怎么做到这一点呢？`strace` 只能记录系统调用，不能拦截。以及，我们希望 Sentry 也能够拦截 trap/fault 等。

## Sentry与应用的交互与隔离 : Ptrace

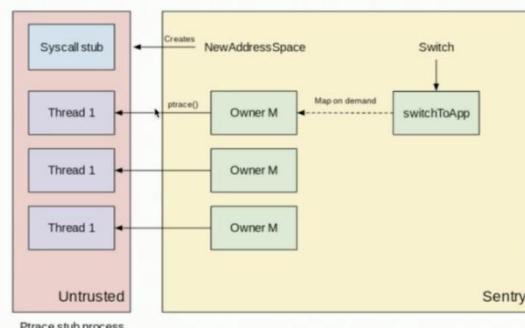
- Ptrace
  - 拦截系统调用和Faults
  - 拦截的事件会通过Signal通知另一个进程
  - 每一次拦截会涉及多次Host OS的上下文切换
  - 兼容现有OS



Ptrace 其实就是一个 signal 的方式。比如应用跑的时候调用 `syscall`，我们做一个 `notify`，把 `wait4` 执行一下。拦截之后，我们通过 `signal` 做一些事情，比如我们要调用 `wait4` 的时候，我们直接调用用户态的 `wait4` 的实现。但是这个会涉及到多次上下文的切换。

## Sentry与应用的交互与隔离 : Ptrace

- 基于Ptrace的Sentry架构



Ptrace 在实现的接口上可能和 strace 类似。通过 Host OS 告诉 owner 去处理一下。这是第一种方法，好处就是没有引入额外的层次，没有新的虚拟化层次。两个都在用户态的权限级别。

## KVM

### Sentry与应用的交互与隔离：KVM

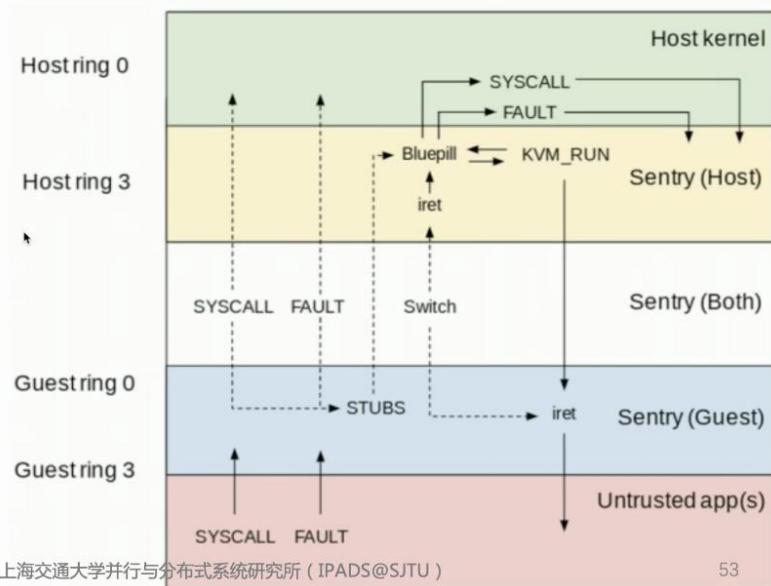
- **KVM**

- 主流的虚拟机监控器，Linux自带
- Sentry会同时运行在Host Ring-3和Guest Ring-0
- 并且在两个特权级之间动态切换
- 应用运行在Guest Ring-3，Sentry可以非常轻易地通过
  - Sysret/iret: 从Sentry返回到用户态应用
  - 拦截Syscall和Fault ( Guest Ring-3下陷到Guest Ring-0 )

KVM 相对于 Ptrace 要有 Linux 相对的支持。在 KVM 这个层次下，Sentry 不是一个单纯的和用户进程在同一个权限级的进程了，它既在 Host Ring-3，也在 Guest Ring-0。应用运行在 Guest Ring-3，因为有这么一个虚拟化支持。Guest Ring-0 的特权级别是高于 Guest Ring-3 的，此时 Sentry 就像是一个 Host Kernel 了。这种情况下做隔离是非常好做的。在正常的虚拟机里面，虚拟机应用和虚拟机内核也是有比较强的隔离的。

### Sentry与应用的交互与隔离：KVM

- **基于KVM的 Sentry架构**

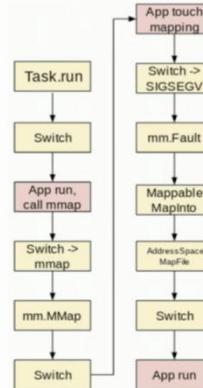


Sentry 可以认为是 Kernel 视角下的一个正常应用，在 APP 这里考虑的话，它又相当于一个有权限的 Kernel。它可以很容易地捕捉到 Syscall 和 Fault，它也可以作为应用调用 Syscall 和 Fault。

Bluepill 出处黑客帝国，很多人觉得这里面可以挖的地方。Bluepill 就是吃到它就可以装作什么事情都没有回到原处。也就是调用 syscall 和 fault 最终还是可以回到 sentry 里面。

## Sentry: 结合Host管理物理资源

- Sentry通过一个MemoryManager的结构来抽象掉OS的内存管理模块
  - 通过CLONE\_VM创建的任务都有自己的MemoryManager
  - 管理任务的VMA
- VMA会在mmap()时创建，但是真正的映射会在使用时建立：on-demand paging
  - 当返回SIGSEGV时处理内存Fault
- Sentry调用Host来分配（或者释放）真正的物理内存资源

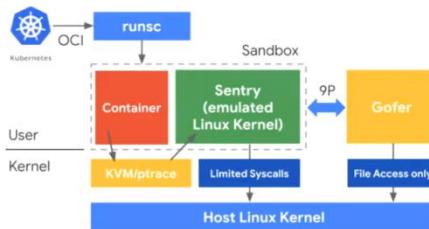


Sentry 在这个基础上就是结合 Host 去管理资源。原来使用虚拟机都是 Linux 和 Windows，这里的虚拟机比较特殊，是一个定制的内核。捕捉 syscall 判断是自己处理还是丢给 Host OS 处理。Sentry 搞了一个 memory manager，把 OS 内存管理的功能架空了，Sentry 把 VMA 拿来自己管理。刚开始跑的时候，我们 call mmap 还是要掉入 kernel 的。后来 APP touch 了 mmap，就会发生 SIGSEGV，这时候还是要掉到 Host Kernel 里分配真正的物理资源。但是一旦物理页资源分配完成了，之后的访问 Host Kernel 就不需要负责了。那这种情况我们就去找 Kernel 去做，但是一旦资源分配完成了有的事情可以 Sentry 做。比如给页表加上只读 bit，一旦后面有了 SIGSEGV，那么 Sentry 就可以接管。

这可以认为是控制面和数据面分离的方法，初始的时候 Host Kernel 要做基础的分配工作。因为 Host 是拿着物理资源的，后来就 Sentry 来管理具体的物理资源怎么使用。到这里 KVM 就很清晰了。其实就是一个裁剪过的 Guest Kernel，有一些功能是没有实现的。中间做了一层抽象防止应用直接接触 Host Kernel。

## gVisor与Docker容器生态的整合

- gVisor兼容了OCI接口，将自己实现成了一个新的容器运行时：runsc



gVisor 也得兼容 OCI 接口，它实现了一个 runsc 运行时，在 runsc 下面有 container 和 Sentry，支持 docker 相关的接口。对下就是使用自己的这一套机制来做虚拟化。这样就让 gVisor 具备了比较强的兼容性，支持虚拟化就可以用 KVM，不支持就可以用 ptrace。这是轻量级虚拟化里比较有代表性的工作。其实就是通过 Sentry 的转发形成了防御纵深的概念。

## ► gVisor与Docker容器生态的整合

- gVisor兼容了OCI接口，将自己实现成了一个新的容器  
运行时：runsc

Running a container

Now run your container using the `runsc` runtime:

```
docker run --runtime=runsc --rm hello-world
```

You can also run a terminal to explore the container.

```
docker run --runtime=runsc --rm -it ubuntu /bin/bash
```

Many docker options are compatible with gVisor, try them out. Here is an example:

```
docker run --runtime=runsc --rm --link backend:database -v ~/bin:/tools:ro -p 8080:80 --cpus=0.5 -it busybox telnet towel.blinkenlights.nl
```

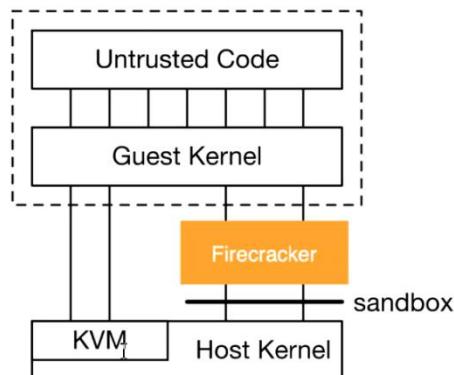
我们可以跑一下试试看，可以使用 gVisor 提供的 runsc 去跑一些容器。

## FireCracker

Amazon 的 FireCracker 和 gVisor 有一点相似。

### FireCracker 🔥

- Amazon开源的轻量化VM系统
- 主要场景：
  - 为基于容器的服务（如无服务器函数）提供轻量化隔离



它其实就是一个轻量化的 VM 系统，方法非常直接。FireCracker 就是一个虚拟机监控器。主要场景就是面向 serverless 的。

**2022/4/8**

今天讲的是大家比较关心的内容，也就是 K8s。因为我们不要求大家从 docker 的底层开始做起，大家可以用 docker 命令创建容器，无非就是在此之上管理它。LAB 要求里有 K8s 的 pod、replica，都会去讲。

## 回顾：OCI与RUNC



- OCI目前主要负责维护：

- 容器运行时标准：将具体的隔离方式从容器中解耦出来
- 容器运行时参考实现 ( reference runtime )
- 容器镜像格式标准

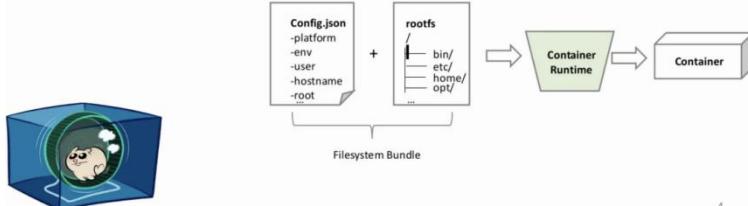


docker 对外的是 docker 的接口，里面有 containerd 来管理镜像，再里面就是 runc，也就是容器的实现标准。只要满足这个标准的，我们都可以叫做是容器，这样就不再限制后端的实现的。

katacontainer 是 runv，gVisor 是 runsc。都是兼容 OCI 标准，这样 docker 就可以运行在不同的运行时上。当然镜像也有一些标准和规约。

## 回顾：OCI与RUNC

- 基于Linux cgroup/namespace等实现的OCI容器运行时
- 作为样例实现，是现在Docker、K8s等的默认容器运行时

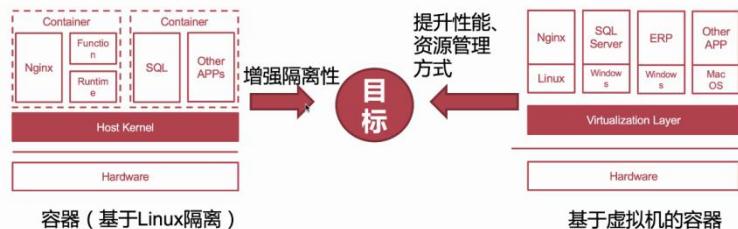


4

现在 runc 用的比较多，它作为样例是 docker 和 K8s 的默认容器运行时，它包含了 rootfs，在我们启动 docker 以后，根目录和 Linux 根目录没有什么区别。

## 回顾：轻量级VM

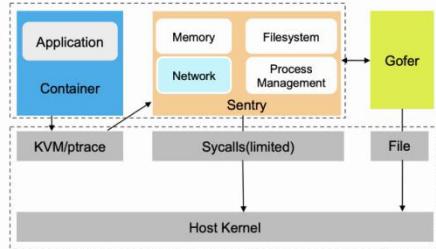
- 目标：容器的性能、易用，但是更强的隔离性



在容器的视角来看，虚拟机和容器各有各的问题，比如性能的问题和隔离的问题。我们的目标就是找到一个既有较好的隔离性，又有比较好的性能。总的来说，确实是因为我们在云环境中有了新需求，所以就往这个方向发展了。gVisor 的思路就是在容器的基础上增强隔离性。

## 回顾 : gVisor

- **Sentry: 定制化的客户机内核**
- **KVM/Ptrace**
- **怎么平衡安全和性能 ?**

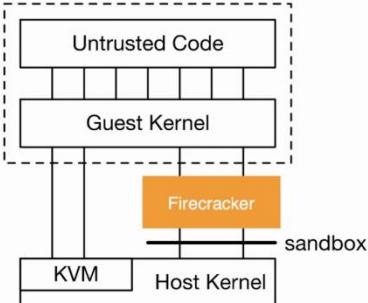


gVisor 其实就是一个客户机内核，这样就可以拦截大部分系统调用自己处理一部分，把一份系统调用交给 Host Kernel 做。在 Sentry 这个角度，我们还是从容器调用到 Sentry，再掉到 Host Kernel，我们加了一层间接层做了一些检查，这样就等于把防御的深度给拉深了。

## FireCracker

### FireCracker 🔥

- **Amazon开源的轻量化VM系统**
- **主要场景：**
  - 为基于容器的服务（如无服务器函数）提供轻量化隔离

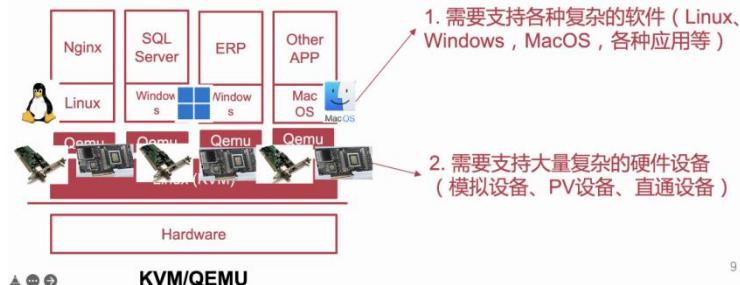


一个比较大的 motivation，就是 Amazon 打算做 serverless（服务器无感知）。就是因为有了 serverless 场景，所以它们开始考虑 FireCracker。轻量化比起虚拟机，它的功能肯定是要下降的，我们不得不裁剪一些功能，比如“调试”这个功能。对开发者来说，调试是很重要的，比如 IDE 的调试比较完善，可以比较好地以 UI 的形式显示出来。轻量级化了以后，可能这些工具都得去掉，对开发有一些不便。

Serverless 的思路就是每次来一个函数，我们就创建一个容器/虚拟机让它运行。不管是虚拟机还是容器，创建都是需要花时间的（虚拟机秒级，容器亚秒级）。在 Amazon 上启动一个虚拟机是相当花时间的，所以 FireCracker 就是希望做轻量化的虚拟机来提升性能。在 FireCracker 跑的应用也是比较轻量级的，可能并不是一套全家桶在运行。

## FireCracker对轻量VM的理解

### • 为什么现有的VM方案比较笨重？



9

业界对 Java 的吐槽就是代码太容错。Spark 上写 MapReduce 代码只需要 2 行，Scala 的简化语法非常地好，而 Java 一般大家都认为代码比较冗长。像 ERP 这种，很多都是对 Windows 开发的。有 TO C 的公司和 TO B 的公司，TO C 的公司因为很多客户用的是 Windows，所以很多软件都是基于 Windows 的。Windows 对于产业界还是比较重要的 workload。这样我们做虚拟化就需要支持 Windows 和 Windows 上的应用。

对下要支持更多的硬件。模拟设备就是用模拟的方法支持设备，直通就是直接把设备给到上面的来使用。直通比较重要的例子就是网卡，如果走正常虚拟化的话，那么就是网卡发给 Linux，它分包给不同的虚拟机。虚拟机可能要配置一个虚拟的网卡来接收网络包。直通就是 Linux 把网卡直接给到虚拟机，虚拟机需要支持网卡从硬件发数据给我，也就是至少也要支持网卡的驱动。

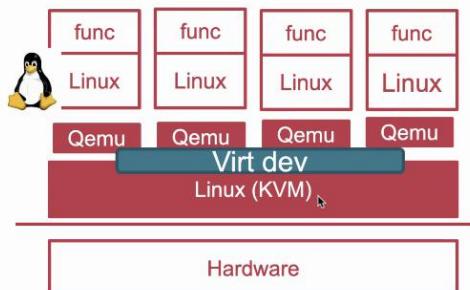
### • 为什么现有的VM方案比较笨重？



10

GPU 的直通也是比较重要的情况。这样我们通过直通，对 GPU 要有相应的支持，这样我们要支持的硬件设备也必然会越来越多。

最早的 ZEN 这样的虚拟机，我们不需要在上做什么事情。上面可能就跑一个 Nginx，虚拟化这一层很简单，比如模拟网卡、IO、地址翻译等。



面向具体场景（固定的软硬件需求），**定制化VM设计！**

1. 定制化Guest Kernel (只实现必要的接口)
2. 定制化虚拟机监控器 (只支持特定虚拟设备)



## KVM/QEMU

11

我们可以定制化 Guest Kernel 和 VMM（虚拟机监控器）。现在只支持虚拟设备，这样我们的代码量就减少了。这其实就是在 Guest Kernel 和 VMM 都比较复杂的今天，采取定制化的思路。FireCracker 没有摆脱虚拟机的设计，这样一个环境下，其实我们虚拟机之间隔离性是比较强的，理论上会比 gVisor 更强。gVisor 本质上就是共享的 Host Kernel，多了一层 Sentry 保护，和虚拟机不太一样。而在虚拟化的场景下，我们就可以认为 Linux/Windows 里有自己的处理方式，可以自行管理。这样 FireCracker 理论上是比 gVisor 更强的。在这个模式下，大家有一定的相似性，虽然没有 share 的 host kernel，但是有 shared Host OS，只是共享的程度小一些。

## FireCracker的目标：安全性

### • 安全与隔离性

- 不同的实例之间提供较强的隔离（理论上比gVisor会更强）
- 能够抵御privilege escalation, information disclosure, covert channels等公有云上比较重要的安全问题

### • 兼容性

- 支持基于Linux的任意应用（二进制和库）
- 主要场景：Serverless（服务器无感知计算）中的函数实例

AWS将FireCracker用于Serverless，  
后续的课程中会进一步介绍。



12

FireCracker 可以抵御一些问题，比如提权。因为我们的接口限制比较多，所以不容易攻击 Host Kernel。信息泄露也类似，自己被控制的情况下也很难操控别人的虚拟机。`convert` channel 就是根据硬件的 bug 来进行攻击，那么我们可以通过虚拟化硬件的方式来进行防御。它继承了虚拟机本身的安全方面的一些好处。只要我们定制的好，我们就可以支持基于 Linux 的任何应用。

在 Linux 上的二进制文件是有固定格式的（elf），我们执行的时候解析转化为虚拟机空间中的映射，而裁剪过的虚拟机中这部分的机制不变，所以程序还是都可以跑的。

## FireCracker的目标：安全性

- 安全与隔离性

- 不同的实例之间提供较强的隔离（理论上比gVisor会更强）

- 基于VM的隔离+一些安全增强技术**

channels等公有云上比较重要的安全问题

- 兼容性

- 支持Guest Kernel 基于Linux裁剪

主要场景：Serverless（亚马逊云服务）下运行的实例

AWS将FireCracker用于Serverless，  
后续的课程中会进一步介绍。



在 Serverless 上，AWS 去调查了相应安全问题，相应地可以做一些隔离技术。

## FireCracker的目标：性能

- 高实例密度

- 在单个机器上跑1000个VM实例！

- 性能

- 实例应用在VM中的性能应该和在Host上是接近的

- 实例之间能够实现性能隔离

- 快速的启动和销毁一个实例（来自Serverless的特有需求）

- 资源管理：动态按需分配

- 对于CPU、内存等资源，在使用时再分配，而不是直接根据给定的资源要求分配

对于云服务商来说，不希望性能差别很大。然后实例之间还有隔离的要求，比如同时跑两个 FireCracker 的实例，它们之间的性能不应该有太大的扰动。在公有云里，不一定能做到动态按需分配。阿里云的内存是不超售的，否则公有云的声誉可能会受到影响。但是对于 Amazon 自己搭建平台的时候，为了更高的资源效率是可以做再分配的。

## FireCracker VMM

### FireCracker VMM：定制化裁剪VMM

- 使用简化的设备模型

- 只支持有限的模拟设备

- 网络和块设备

- 串口

- Partial i8042 (PS/2 keyboard controller)

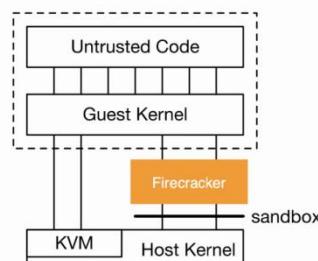
- no BIOS, no PCI, etc

- Virtio for network and block devices

- 1400 lines of Rust

- 效果：相比Qemu，能够减少

- 96%代码行数！



设备本身是 VMM/Linux 变得这么臃肿的原因。Virtio 就是 guest kernel 和 VMM 之间的通

信。Rust 支持比较强的语言级隔离，相对来说是比较安全的语言。

在安全增强方面，是根据最新的攻击来做的：

## FireCracker VMM : 安全增强

- FireCracker在VM隔离的基础上，进一步增强了安全性
- 抵御最前沿的硬件攻击（如Spectre，Meltdown）
  - 关闭SMT特性
  - Host kernel上应用大量相关补丁：KPTI, Indirect Branch Prediction Barriers, 等
  - Host Kernel上打开对应的安全选项：Speculative Store Bypass mitigations等
- 额外的沙箱保护：Jailer / Sandbox
  - VM之间会被PID、Network等命名空间隔开
  - 使用Seccomp等技术限制VMM和VM能执行的系统调用

可以通过一些选项防御最前沿的硬件攻击，还可以在软件层上做一些保护。还可以通过Seccomp 限制能够使用的系统调用的数量。

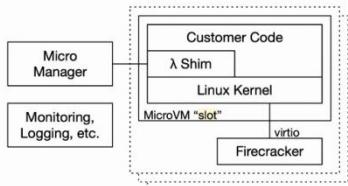
## FireCracker VMM: 动态资源

- FireCracker中没有使用Linux cgroup（控制组）对资源进行管控
- 而是使用了：**Rate Limiters**
  - 统计VM使用的资源上限，并进行限制
  - 相比控制组，Rate limiter更加简单，也失去了很多灵活性
  - 对于FireCracker面向的场景已经足够

VMM 动态资源其实用的是 Rate Limiter。比如 FireCracker 的一台虚拟机跑在一台物理机上，这样虚拟机就可以独占整个设备去跑，cgroup 可以做到这一点，但是 Rate Limiter 是不够的。在公有云的场景下，资源上限是我们花钱买来的，所以不需要上 cgroup 这样的控制组。

## FireCracker MicroVM : 定制化裁剪的客户机

- MicroVM:
  - FireCracker中运行应用的最小单元（一个定制化的VM）
  - 最小化的Linux内核加上和业务定制化的用户态环境
  - 内部运行少量的控制进程，和外部进行同步
- 效果：能够支持~1000个实例



首先明确一下概念，MicroVM 是允许应用的最小单元，里面就不会出现新的单元了。因为它是定制化裁剪过的，所以它的 Linux 内核是最小化的，再加上一个定制化的用户态环境。一开始容器的定制化环境也是定制化出来的，这里就是把内核的环境也做成了一个最小化的

环境。在云环境里，我们的 FireCracker 总是要管理虚拟机，还需要资源分配等，需要和 Manager 进行同步。效果就是支持同一台机器上的 1000 个实例，在一些测试里 MicroVM 的启动时间可能还比容器快。定制化的问题就是原先认为理所当然的东西会没有。

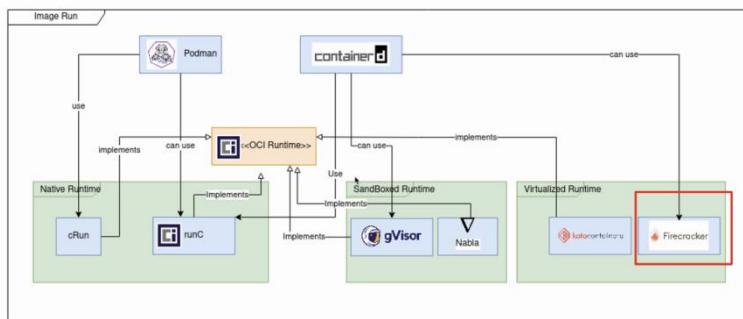
## › FireCracker和gVisor的异同

Firecracker	gVisor
Relies on guest and host kernel	Relies on Sentry and host kernel
Narrow syscall interface	Wider syscall interface
Low memory footprint	Low memory footprint
Type safe language (Rust)	Type safe language (Golang)

### 两种方案都通过限制与Host内核的交互，来提升安全性

都是两层隔离，gVisor 就是 Sentry 隔离，而 FireCracker 就是基于虚拟化的隔离。Sentry 支持的系统调用比较多，而 FireCracker 因为使用了 Seccomp，导致可以使用的系统调用更少。两个都用了类型安全的语言。

## › FireCracker与Docker容器生态的整合



因为 containerd 是 docker 的镜像的支持。

## 总结：轻量级VM虚拟化

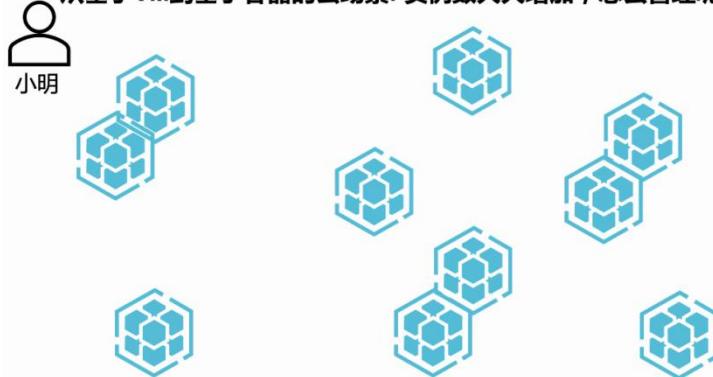
- 兼容Linux容器的高效和VM的高隔离性
- 典型系统：gVisor、FireCracker
  - gVisor: 将System API进行隔离，但是仍然复用Host kernel资源管理
  - FireCracker: 定制化VMM和客户机内核，实现面向特定场景的VM轻量化
- 都兼容OCI运行时规范，都支持容器镜像
- 能够和Docker、K8s生态整合

## K8s

为什么有编排的概念呢？

## 容器编排 (Orchestration)

从基于VM到基于容器的云场景: 实例数大大增加, 怎么管理呢?



在交响乐中, 编排整个组合是很复杂的事情, 要精心排练才可以得到合奏。而这里就是把很多 parts 协调在一起。从虚拟机到容器这个云场景下, 我们的实例数大大增加, 管理起来就更麻烦了。

## 容器编排 (Orchestration)系统

在(通常)大规模环境中管理容器的生命周期的系统, 如云、数据中心等

- 容器编排系统需要支持:

- 容器的配置、部署、服务供应等
- Availability
- Scaling
- 调度
- 容器应用版本更新
- 健康检查 (Health checks)

配置、部署以前都是用户手动启动的, 现在就是编排系统自动帮我们做。有关系的容器可能就通过调度放在一起。K8s 就是这么一个容器编排的引擎,

## Kubernetes

- “Kubernetes is an open source **container orchestration engine** for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation.” – Kubernetes docs



## K8s对于云原生等场景十分重要

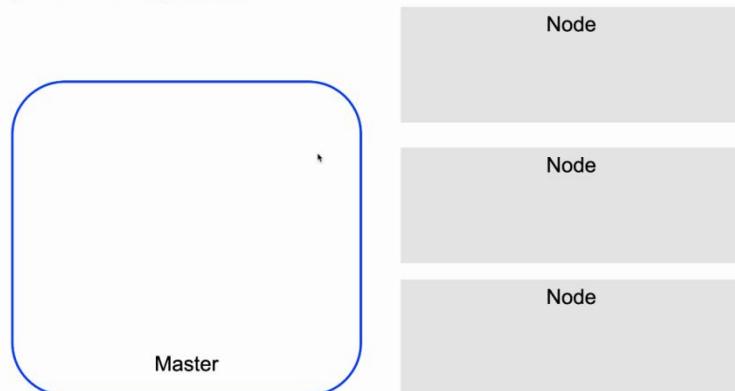
- K8s被认为是数据中心和云上新的OS
  - “Cluster OS”
- 云计算平台, 如Serverless计算平台通常依赖于K8s管理容器和物理资源
- 负责容器的编排调度

逐渐大家发现 K8s 对于云原生的场景非常重要，在云环境中容器就是应用，那么 K8s 就可以认为是云的操作系统。既然要上这门课，我们就得知道它是怎么实现的。实际上它主要负责容器的编排调度，底层的硬件资源的管理可能也是可以归 K8s 管理的。

## K8s 的架构和组件

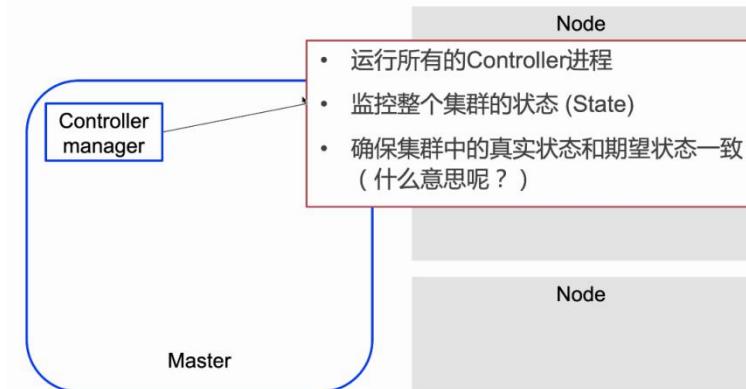
不要求同学都按照今天的内容来实现。

### Kubernetes集群: Master + Nodes (workers)架构



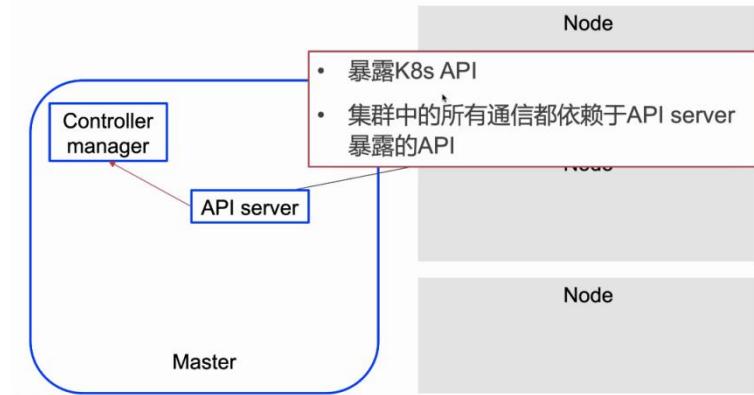
它是经典的 Master/Worker 架构。

### K8s组件: Master



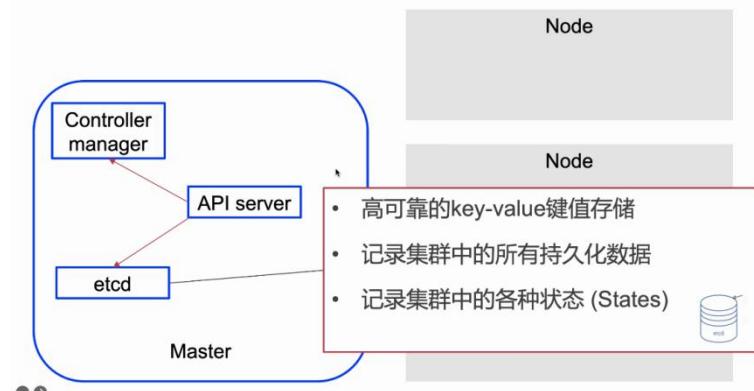
每个 worker 会定义 state，它需要监控并保证真实状态和期望状态一直。

## K8s组件: Master



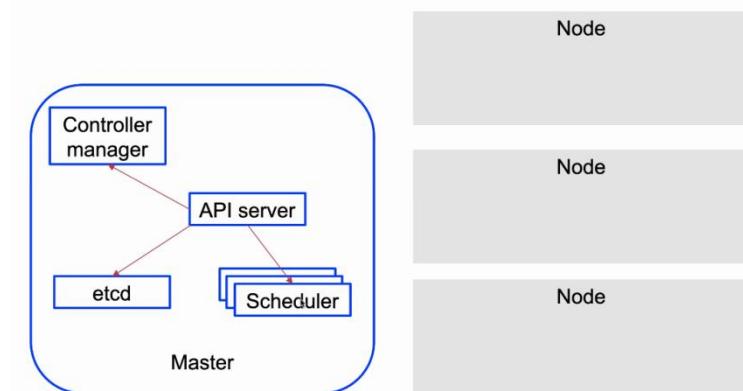
API 也是 K8s 定义的，有一些容器管理的功能需要暴露出来。

## K8s组件: Master



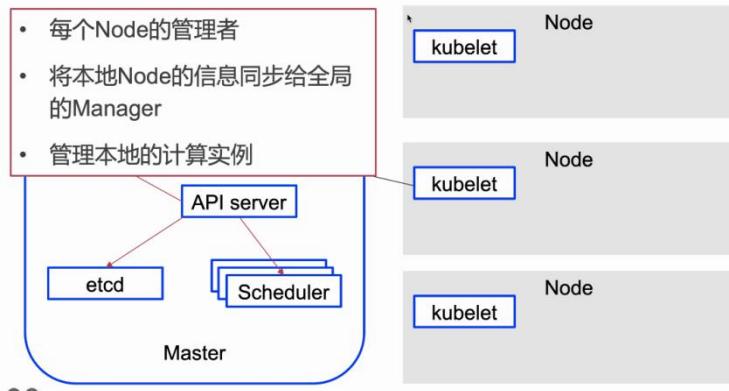
etcd 是类似于 raft 的实现，可以保证 consensus。即使有机器挂掉，我们也可以恢复

## K8s组件: Master



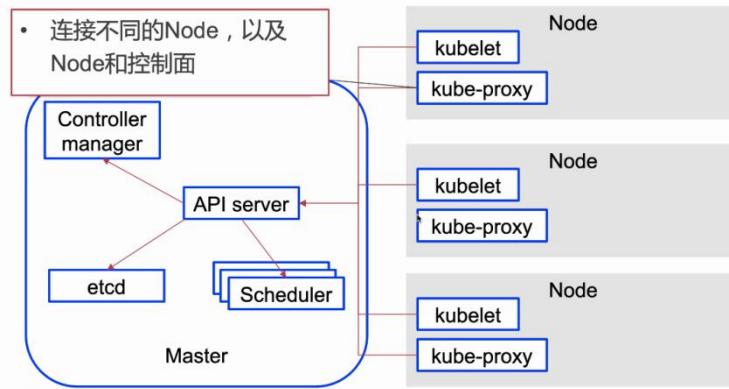
scheduler 就是做调度。

## K8s组件: Node



kubelet 就是每个 worker 本地的管理者，管理本地容器。

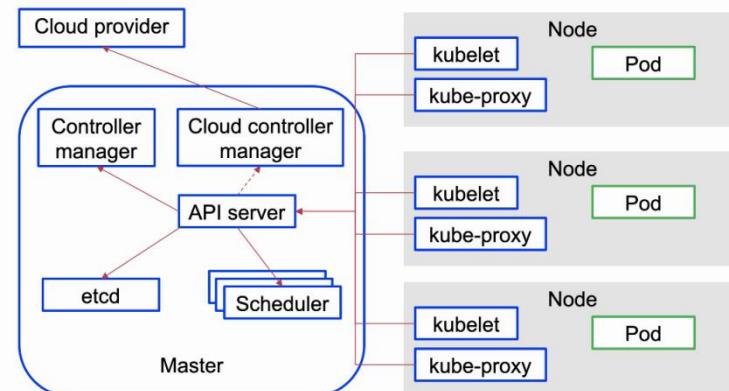
## K8s组件: Node



node 和 node 之间也需要有通信，这样我们就需要走 proxy。proxy 还需要和 master 连接。

## K8s Pod

## K8s计算单元 : Pod



最后是我们的 pod，它是我们的应用实例。每个 Node 有一个或多个。

# 为什么要有Pod？

- 容器抽象：

- 无关的进程不推荐放在同一个容器中，最好一个容器只有一个进程和它的子进程

- 应用的新需求：

- 有的应用程序需要支持同时存在的的辅助进程
  - 比如，应用自带的Proxy进程、记录应用的日志或监控数据的进程等
  - 辅助进程和应用存在一定的绑定关系，但是并不紧密 → 放在两个容器中

容器本身可能是一个轻量级的单元，最好只有一个进程和子进程。我们希望运行环境比较干净，无关进程不要放在一个容器里。辅助进程和应用不一定要在同一个容器里，比如 proxy 进程就可以和容器分开。在分配资源的时候，proxy 可以占用比较少的资源。

# 为什么要有Pod？

- 容器的局限性：

- 需要多个容器来维护松耦合的绑定关系
  - 在容器编排时提高了复杂度
    - 调度时这些容器是不是在一个节点？
    - 扩缩容是否一起进行？
    - 如何通信？通信开销？
    - .....

在这样的情况下，单一容器抽象就可能不够了。但是以多容器暴露给上面可能就复杂，比如我们调度的时候，这些容器要不要调度到同一个节点。扩缩容的时候，是否在同一台机器上扩增？

显然容器和容器之间是有比较强的绑定关系（松耦合），比如我们容器启动的时候，必须启动 proxy 容器。在这种情况下，我们应该把它们合在一起。

## Pod：Kubernetes的最小部署单位

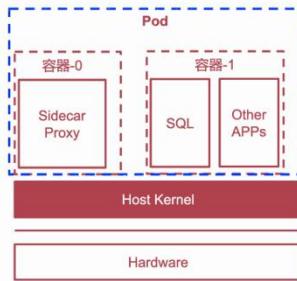
- Pods（豌豆荚）：Kubernetes 中创建和管理的、最小的可部署的计算单元

- 容器抽象的扩展，由一个或多个容器组成
  - 可以天然共享网络和存储
    - 网络：Pod中容器共享Network命名空间
    - 存储：通过设置共享的存储卷（Volume）来共享数据



我们在容器的抽象基础上搭了一个更高的结构。在 Pod 的基础上，我们可以去构建网络和存储方面的抽象。比如这些容器可以共享网络命名空间。存储是通过共享的 volume 来实现数据卷的共享。

## 容器 v.s. Pod



容器是更里一层的概念。容器 0 是做网络连接的，而容器 1 是我们的 APP。它俩之间有比较强的网络关系，所以我们就用 Pod 包起来。

## K8s对Pod的管理

### • Pod的创建：

- 通过配置文件指定Pod中各个容器的镜像等信息
- kubectl apply -f config.yaml来创建相关的pod

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
```

config.yaml

数组，可以有多个容器  
容器镜像  
暴露的端口

接下来我们来讲 Pod 的管理。我们需要把用户写的 config 文件转化成相应的 pod。比如我们的名字叫什么，spec 里有我们的容器 containers：指定每个容器镜像和暴露端口。

## Pod 的生命周期

### Pod的生命周期

#### • Pod的生命周期：

1. Pod被创建，并赋予一个UID
2. 被调度到某一个节点上，并在终止之前一直运行在该节点
3. 节点的fail、资源突然不足等，都可能导致Pod被终止

OS 的时候有进程的生命周期，而 Pod 也是类似的。不恰当的比喻就是 Pod 是进程而容器是线程。Pod 创建的时候被赋予一个 ID，在创建以后有一个调度执行的过程，在终止之前我们必须一直运行在该节点。

## Pod的生命周期

- **Pod包括以下状态：**

- Pending : Pod已经被k8s集群接受，但是容器没有全部被创建，如下载镜像的时候或者集群资源不够的时候
- Running : 已经部署到某个节点，所有容器被创建
- Succeeded : 所有容器都成功终止且不再重启
- Failed : Pod中至少有一个容器以非 0 状态退出或者被系统终止
- Unknown : 因为某些原因无法取得 Pod 的状态，通常是因为与 Pod 所在主机通信失败。

Pending 就是比较初始的状态，刚刚接受还是还没有创建好，这算是一个中间状态。

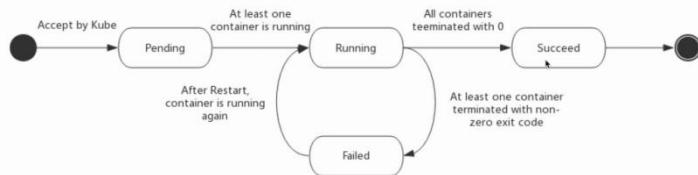
Running 就是容器可以运行了。

Succeeded 就是容器成功终止。

Unknown 状态，K8s 无非就是通过 master 和 kubelet 来判断容器的状态，如果网络不响应，那么我们是观测不到 pod 的状态的，那么就是 Unknown 状态。

- **Pod包括以下状态：**

- Pending, Running, Failed, Succeed, Unknown



Running 状态如果容器以非 0 终结，可能就到 Failed 状态了。这个图有点像我们编译原理学的状态机。我们在管理的过程中，我们认为 Failed 了以后一定要重做任务，而在进程里我们没有规定成功和失败，这是因为在云环境中有更强的应用语义，最终我们希望达到成功的状态。

Fail 之后，K8s 是有责任去做一些事情来恢复 Pod 的。

## Replication Controller

### Replication Controller (RC)

- **RC对应的功能：保证集群中指定Pod副本数量**

- 多个或者1个

- **两种处理方式：**

- 当对应的Pod数量少于指定数目时，RC自动启动新的Pod副本
  - 当对应的Pod数量多于指定数目时，RC自动杀死多余的副本

- **场景：当通过RC运行指定的Pod，并配置为1时**

- 即使出现错误导致Pod退出，RC也会立即重启一个Pod实例
  - 高可用

- **RC是K8s中，较早引入的保证Pod高可用的API对象**

一个比较常见的配置就是配置为 1，如果有一个错误导致 Pod 退出了，那么我们会重启，这也是可以达到更好的 availability 的。

比如我们发现最近的流量超过了 100，这时候就引入了 Load Monitor，就可以扩容，这就提高了可扩展性。

## API 对象

什么是API对象？为什么需要API对象？

容错只是K8s提供的一个功能点，为了支持各种各样的复杂功能，K8s用API对象来管理各种集群上的能力

老王

## API对象

- K8s集群中的管理操作单元
  - 引入新的功能或特性时，一定会引入对应的API对象
- 使用（或配置）K8s的某个功能是通过操作API对象实现的

大家可以类比成Unix/Linux系统中，“所有”的系统抽象都是File

老王

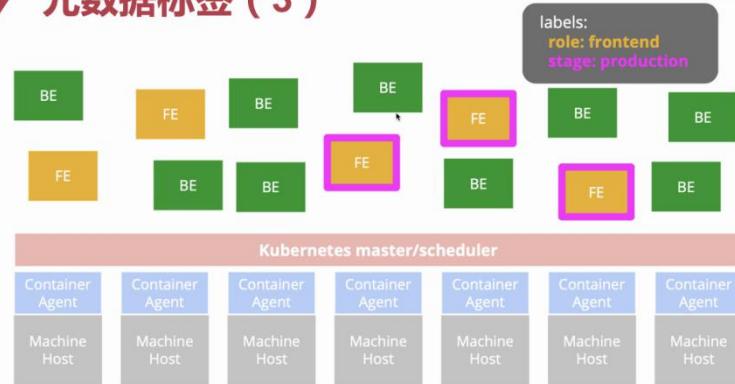
我们把功能抽象成了 API 对象，它包含三种属性。

## API对象的属性：元数据

- 元数据 (metadata)
  - 用来标识 API 对象
  - 至少有 3 个元数据：namespace，name 和 uid；
  - 灵活的标签 (labels) 用来标识和匹配不同的对象
- 标签的灵活用法
  - 案例：用户可以用标签 env 来标识区分不同的服务部署环境
  - 分别用 env=dev、env=testing、env=production 来标识开发、测试、生产的不同服务。
  - 分别用 env=FE、env=BE 来标识前端和后端服务。

标签是用来表示不同的对象的。不同的部分需要的服务是不一样的，比如开发部分我们需要 debug 环境，testing 可能需要测试用例，而 production 可能就需要生产的服务。

## ▶ 元数据标签 (3)



## API对象的属性 (2) : 规范 (Spec)

- 规范描述了：
  - 用户期望 K8s 集群中系统达到的理想状态(Desired State)
- 例如：
  - 用户可以通过Replication Controller设置期望的Pod副本数为 3
- 规范的具体内容和API对象相关
- K8s中所有的配置都是通过API对象的规范去设置的

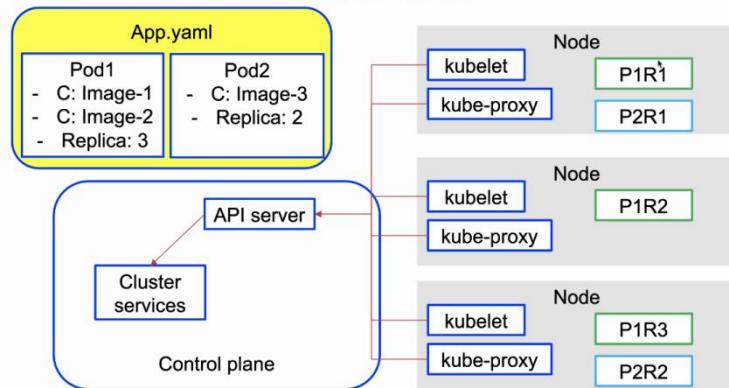
API 对象是和我们的期望值有关系的。

## API对象的属性 (3) : 状态

- 状态 (Status) 描述了系统实际当前达到的实际状态
- 规范定义了期望状态：
  - 用户可以通过Replication Controller设置期望的Pod副本数为 3
- 如果当前实际的Pod副本数为1，那么其实际状态为：
  - Pod副本数为 1

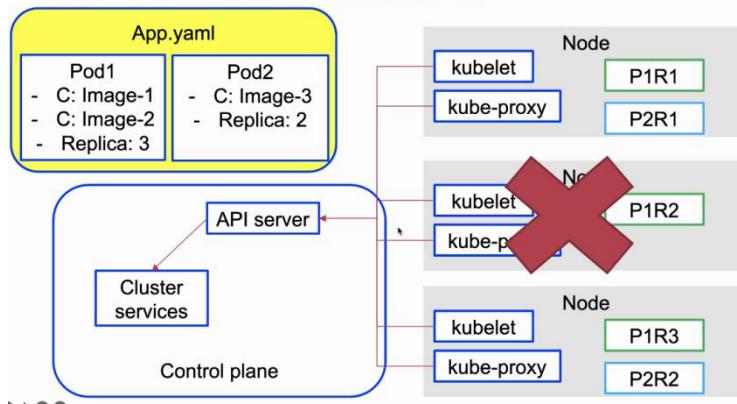
K8s 里可以实时观测真实状态，然后和我们的期望状态做比较，如果不够了我们就去扩容。

## ▶ 基于期望状态的集群管理



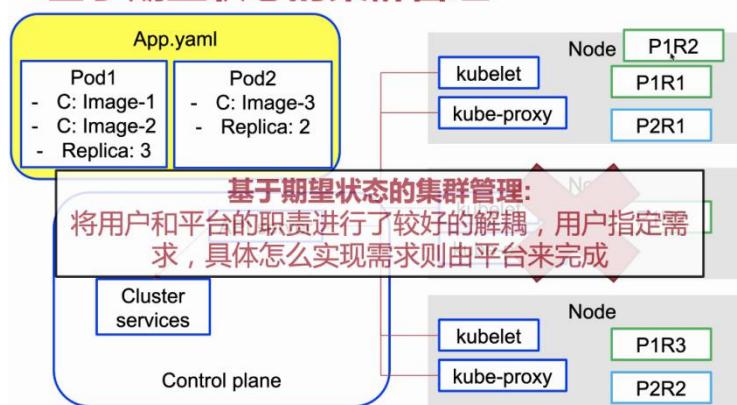
通过 API Server 暴露了 replica 的信息，通过 kubelet 和 API Server 我们可以实时得到当前集群中的节点状态

## ▶ 基于期望状态的集群管理



如果此时 Node2 crash 了，我们的 API Server 检测到了状态不匹配就会在 Node1 创建了一个新的 P1R1 的 Replica。

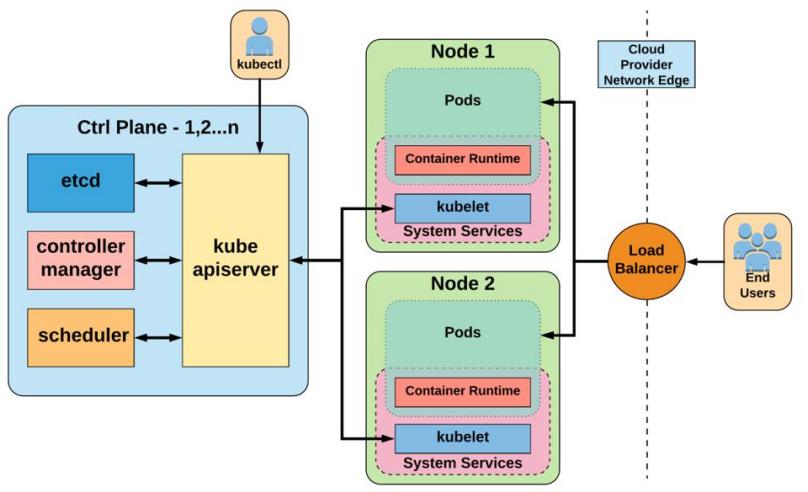
## ▶ 基于期望状态的集群管理



是否是一个好方法是要看用户的具体需求的。扩容和创建 replica 是 K8s 自己实现的，只是提供给用户指定需求的 API，底下 K8s 必须支持备份和扩容。这是一个比较好的解耦，用户需要考虑的事情是比较少的。

2022/4/12

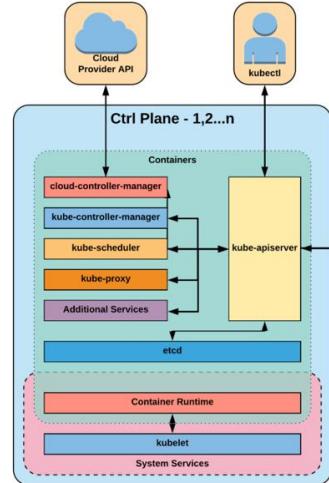
## 回顾：K8s架构：控制面/Master + Node结构



上次我们介绍了 K8s 的基本架构，主从结构分为 master 和 worker。

## 回顾：K8s架构：控制面/Master

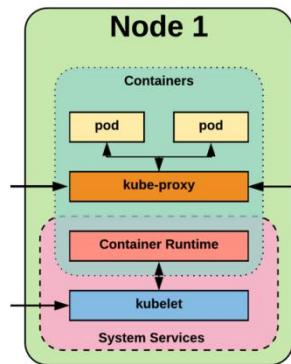
- kube-apiserver
- etcd
- kube-controller-manager
- kube-scheduler



API Server 是提供各种各样的 API 的，API Server 暴露了什么 API，我们才可以提供 API 控制，做一些 K8s 相关的操作。对内还包括了 etcd 这种做共识的模块，控制面的元数据都需要 etcd 这种来保证高可用。Controller Manager 就是来起一些 Controller，来控制从节点。Scheduler 来选择哪一个节点来适合运行我们的任务。

## 回顾：K8s架构: Node

- kubelet
- kube-proxy
- Container Runtime Engine



在从节点这里主要是如上的这个三个对象。

## 回顾：API对象 (或简称对象)

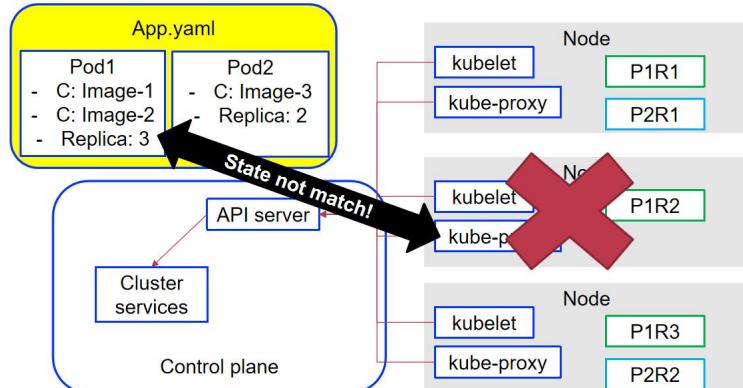
- 对象是K8s集群中的管理操作单元
  - 引入新的功能或特性时，一定会引入对应的API对象
- 使用（或配置）K8s的某个功能是通过操作API对象实现的
- 上节课的例子：
  - Pod: 多容器的API对象
  - Replication Controller (RC): 复杂容错等管理，新一代的K8s中使用ReplicaSet

K8S 里 Object 里是很重要的抽象，在 Unix/Linux 里，万物皆文件，我们通过操作文件来操作硬件，而在 K8S 里就是通过 API 对象来操作功能或使用特性。

上节课我们讲了两种例子，一个是基本的 Pod，它是我们要实现的基本 API 对象，一个 Pod 可以包含多个容器从而做 API 的部署。比如主容器跑我们的网站，另一个容器做负载均衡，再有一个容器做调试框架。

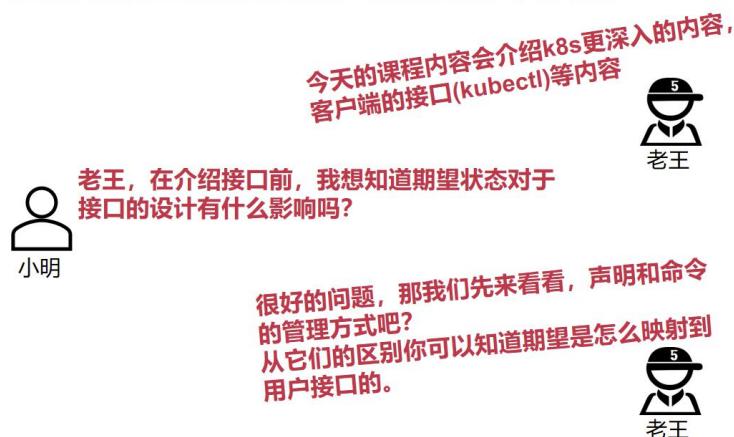
再有一个是 RC，在我们的 minik8s 中有 ReplicaSet 的概念，是最新一代的 K8S 来使用的。

## 回顾：基于期望状态的集群管理



Pod 里的元数据描述了我们期望的状态，如果我们实际的状态和期望状态不匹配，那么我们就需要比如起一个新的容器来符合这个期望状态，比如 Replica 数量等。

## 基于期望的集群管理是怎么来的?



## 声明式和命令式的管理方式

这节课会从用户控制的角度来介绍。在集群管理里，提供了声明式和命令式两种管理方式。这不是指具体的语言特性，而是一个抽象的思想。

### 命令式

#### 命令式 (Imperative)

- 有些场景下，也称为“指令式”
- 即，命令平台如何做某事
- 平台会按照命令实现，但是结果是否符合预期依赖于用户自己的指令是否正确

做出来的菜长啥样全看菜谱对不对



命令式也叫作祈使式，也就是我们命令平台去做一些事情。我们照着菜谱一步步去执行就可以了。在 Github 上有程序员菜谱，全部以定量的方式来描述做菜步骤，像一个脚本告诉我们怎么做，做出来就和菜谱差别不大了。

## 声明式

### 声明式 (Declaritive)

- 有些场景下，也称为“描述式”
- 即，告诉平台你想做什么
- 不需要知道怎么做的，平台会去实现你给的要求
- 平台需要能够自动实现特定目标



给平台原材料和目标的菜，  
不关心怎么做

我们和饭馆老板说要吃什么什么菜，老板就直接去做。我们不用关心中间的步骤。我们提供给平台的是 `what` 需求，平台帮我们解决 `how`。

声明式的典型案例：SQL 语言

```
SELECT * from students
INNER JOIN advisors
WHERE ,
students.advisor_id =
advisors.id
```

```
studentsWithAdvisors = []
Struct student, advisor
For (int i=0; i<students.length; i++) {
    student = students[i];
    for (j=0; j<advisors.length; j++) {
        advisor = advisors[j];
        if (advisor && student.advisor_id == advisor.id) {
            studnetsWithAdvisors.push(j,...);
        }
    }
}
```

图 5 SQL 语句（左） 真正的实现（右）

SQL 的 `query` 只是告诉我们要做什么事情，那么具体怎么做，我们实现可能是一个 C 代码，具体指定了怎么遍历这个数据结构，怎么存储结果。左边就是“做什么”，而右边就是“怎么做”，是因数据结构而异的。

### 为什么我们很少用“声明式”？



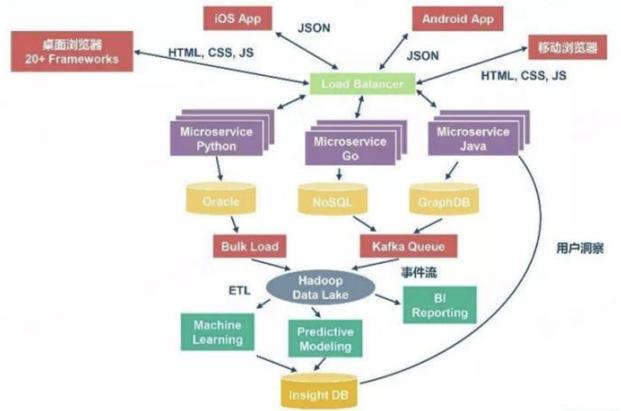
- 程序员喜欢控制事情的发展，不习惯系统中存在脱离占空的自动化处理过程
- 不清楚想要的是什么
- 归纳和提取目标需求不是一件容易的事情

在编译原理中，我们讲过 `lazy evaluation`，我们可以把一个对象存下来，等待不得不用的时候再去求值。求值的部分就被程序语言的运行时来确定我们什么时候需要这个值，以前是程序员强制要求 `read/write`，现在就是运行时自己去控制。

我们用 C 和 C++ 的时候，很少使用声明式。大家现在学 OS 会使用内联汇编，控制事情的发展已经越过编译器了，这样是对我们整个系统进一步的控制。之后可能我们对 OS 做的事情也不满意，需要控制 OS 的一些行为，也就是更低层的东西。所以我们使用声明式可能

没有这么多，并且提取声明式的需求也没用这么容易。在我们一开始写代码的时候，我们是边写边想我们的需求，所以我们还是使用命令式在思考问题，只是最后发现可以使用声明式来做抽象。

## 试着用“命令式”部署这样一个系统



刚才说的更多我们是研究一个点，比如 OS 的 FS 部分，硬件部分。如果我们要理解 OS 的各个部分，对一个人来说有点难度。对于复杂系统，上图是一个微服务的系统，包含移动设备、PC 浏览器、还有不同语言的数据库、分析的机器学习应用等。这样一个复杂系统，对一个人来说需要部署这样一个系统，如果全部手动部署起来，是很复杂的。光说浏览器可能就是 20 多个部件了，所以我们需要更高级的抽象。在 K8S 里，我们肯定是需要声明式的方法去部署让我们把复杂系统组合起来部署。

## 命令式和声明式的比较

### 命令式 v.s. 声明式

- 需要写 step-by-step 的脚本，考虑各种异常情况，依赖开发者个人经验
- 脚本在不同的环境中可能会有不同的处理
- 对于异常情况，比如断电等很难考虑完全
- 多人协作较为困难
- 使用配置文件描述最终状态，不需要考虑中间步骤，易于编写
- 易于理解和维护
- 重复部署不会产生不一致的结果
- 平台负责在不同的环境中实现相同的效果

命令式的脚本很细节，异常情况的考虑也依赖于个人经验。所以命令式可能就适合在 K8S 里测试单个节点做调试的时候，那么我们可能需要脚本来控制机器的运行情况或者模拟一些异常。

而声明式就是在命令式里抽取共同点做出来的平台，平台根据配置文件来得到最终状态是怎么样的，编写起来相对比较容易。它的抽象更高级，因为需要命令式的知识提取。

## ► K8s的核心设计就是围绕声明式的

- 是“期望状态”设计的另一个维度体现

## What is Kubernetes?

This page is an overview of Kubernetes.

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

K8S 还是围绕声明式来实现的。云本身就是一个大的分布式系统，包含机器多、网络复杂。因为我们系统太大了，如果没有声明式的支持，每天程序员的工作量都会耗费在没有意义的东西上。

## Kubectl (K8s 的命令行工具)

- K8s提供的命令行工具，允许用户/开发者和一个K8s集群的控制面(Master)进行交互



其中交互的对象指的是 API 对象。操作包括创建、获取、描述、删除。

例子：

```
kubectl [command] [TYPE] [NAME] [flags]
```

```
kubectl get pods pod1
```

- **get:** 显示一个或多个资源，**pods:** Pod资源/对象，**pod1:** 具体的pod名
- 列出pod1的相关信息（如果不指定pod1，会列出所有的信息）

- **Notes for MiniK8s Lab: 需要实现自己的命令行工具**

- 参考kubectl的手册了解要实现的主要命令：  
<https://kubernetes.io/docs/reference/kubectl/>
- 注意：不是所有的命令都需要实现！需要对接上miniK8s中提供的API

为什么我们要详细讲 kubectl 呢？大家可以参考它实现的命令来设计自己的系统，大家最终是要展示自己的 miniK8S 的，展示的一个好的方法就是我们实现命令行方式，比如写一个 minikubectl。这样展示的时候比较方便，写个 UI 也是可以的。因为我们写 miniK8S 的时候会有一个类似的 API Server，这样就可以对接我们的命令行工具。

## 从 Docker 命令来看 kubectl

Docker 本身可能在调用关系上也有顺序，这里我们有一个简单的 docker run，叫做为 nginx-app 的一个 container。并且我们使用 ps 列出来。

```
$ docker run -d --restart=always -e DOMAIN=cluster --name nginx-app -p 80:80 nginx
```

```
55c103fa129692154a7652490236fee9be47d70a8dd562281ae7d2f9a339a6db
```

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
55c103fa1296	nginx	"nginx -g 'daemon of...'"	9 seconds ago	Up 9 seconds	0.0.0.80->80/tcp	nginx-app

Case from: <https://kubernetes.io/docs/reference/kubectl/docker-cli-to-kubectl/>

\$ docker run -d --restart=always -e DOMAIN=cluster --name nginx-app -p 80:80 nginx
---

55c103fa129692154a7652490236fee9be47d70a8dd562281ae7d2f9a339a6db
--

\$ docker ps
--------------

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES 55c103fa1296 nginx "nginx -g 'daemon of..." 9 seconds ago Up 9 seconds 0.0.0.80->80/tcp nginx-app
---

\$ kubectl create deployment --image=nginx nginx-app
--

deployment.apps/nginx-app created
-----------------------------------

\$ kubectl set env deployment/nginx-app DOMAIN=cluster
--

deployment.apps/nginx-app env updated
---------------------------------------

\$ kubectl expose deployment nginx-app --port=80 --name=nginx-http
--

service "nginx-http" exposed
------------------------------

Case from: <https://kubernetes.io/docs/reference/kubectl/docker-cli-to-kubectl/>

kubectl 也可以实现 docker 的指令，等于在容器抽象上包了一层 deployment 抽象。这个抽象我们这节课不会详细展开。最后我们就 expose，对应了我们的端口配置。这样我们就通过 kubectl 变成了包含左边配置的配置文件。它背后就是执行 docker run 的指令，它就是把 docker 指令包起来做成了新的抽象。

## kubectl 管理对象

Q: Kubectl 似乎核心就是在操作 k8s 的 API 对象，这个和我们前面学习的命令式声明式有关联吗？

A: 大家可能就是想问，怎么看它是命令式还是声明式呢？事实上 kubectl 把管理方式分成两类，命令式和声明式。

## 命令式

- 使用**指令式命令**管理 Kubernetes 对象
- 前面介绍的kubectl的基本命令，就是用**指令的方式**管理K8s的对象的

kubectl create service nodeport <服务名称>

- 比如这个例子，会创建一个“Service”的对象（周五会介绍Service）  
Ref: <https://kubernetes.io/zh/docs/tasks/manage-kubernetes-objects/imperative-command/>

命令式会比较简单，比如我们的 create 一个 service 对象。这里非常清楚地告诉 controller 我们要创建什么样的服务。我们可以看到命令式和声明式是一个相对的概念。我们没有告诉它怎么创建 service，从这个角度来说可能像是声明式。只是说这里的命令式说的是比较清楚的一个指令，就是创建一个服务。

- 使用**配置文件**对 Kubernetes 对象进行命令式管理
  - kubectl 命令行工具 + YAML/JSON 编写的对象配置文件来创建、更新和删除 k8s 对象
  - **创建对象:** kubectl create -f <filename|url>
  - **删除对象:** kubectl delete -f <filename|url>
  - **查看对象:** kubectl get -f <filename|url> -o yaml

也支持使用 YAML 传入配置文件。这也是命令式，主要还是看操作是否是一个比较具体的操作。

## 声明式

- 使用**配置文件**对 Kubernetes 对象进行声明式管理
  - 和命令式配置文件类似，需要指定配置（在目录中）
  - 使用 kubectl apply 来创建指定目录中配置文件所定义的所有对象，除非对应对象已经存在：

kubectl apply -f <目录>/

声明式是更加抽象的，比如 apply 就是一个抽象的概念，我们并不知道它是创建还是删除还是获取，它会严格按照配置文件来实现一些行为。配置文件描述了对象是怎么样的。apply 是看这个目录里是否存在配置文件，如果存在多个，那么我们也创建多个。这里面有容器和容器之间的关系和对象和对象之间的关系，这都在配置文件里描述的比较清楚了。我们的 kubectl 就是把它给实现出来。这里就是比较 high-level 的管理，不是实现一些具体的事情。

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80

```

图 6 YAML 配置文件

这就像我们去饭馆吃饭一样。

## kubectl 管理对象中声明式和命令式的比较

- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>• <b>声明式：</b></li> <li>- 描述目标状态，而不是具体操作</li> <li>- 配置文件中是<b>最终的对象和它们的状态</b></li> <li>- 不关心k8s怎么实现这个状态</li> </ul> <p><code>kubectl apply -f &lt;目录&gt;/</code></p> | <ul style="list-style-type: none"> <li>• <b>命令式：</b></li> <li>- <b>手动去维护对象</b>，创建、删除、查看状态等</li> <li>- 配置文件描述当前的对象</li> <li>- 创建对象: <code>kubectl create -f &lt;filename url&gt;</code></li> <li>- 删除对象: <code>kubectl delete -f &lt;filename url&gt;</code></li> <li>- 查看对象: <code>kubectl get -f &lt;filename url&gt; -o yaml</code></li> </ul> |
|--|--|

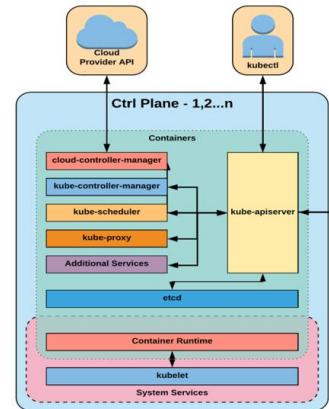
声明式里就是最终的对象和它们的期望状态，K8s 就是通过 `apply` 去达到它最终的对象和状态。声明式就是把最终目标告诉 K8S，让 K8S 去实现。

大家在 LAB 里怎么去用它呢？大家平时可以实现一些简单的 `create` 和 `delete` 操作，可以帮助我们测试并且实现一些小规模的 K8S 里的单元，比如我们 `create` 一个 Pod，其实也可以支持一些声明式的一键部署，比如 `apply` 一下就一键部署了多个 Pod。命令式的扩展性是不如声明式的。

这里我们讲完了命令式和声明式，它们的共同点就是通过配置文件描述对象，大家在做 lab 的时候本身就是要设计配置文件。

## kubectl 和 K8s 集群内部的交互

- **kube-apiserver**
  - 图中, kubectl直接连接apiserver
- **etcd**
- **kube-controller-manager**
- **kube-scheduler**



那 kubectl 是用户接触的一个客户端, 它是怎么连到 K8S 里的呢? 它可以连到 API Server 调用 API, 去操作我们的 etcd、controller、scheduler、proxy 等。

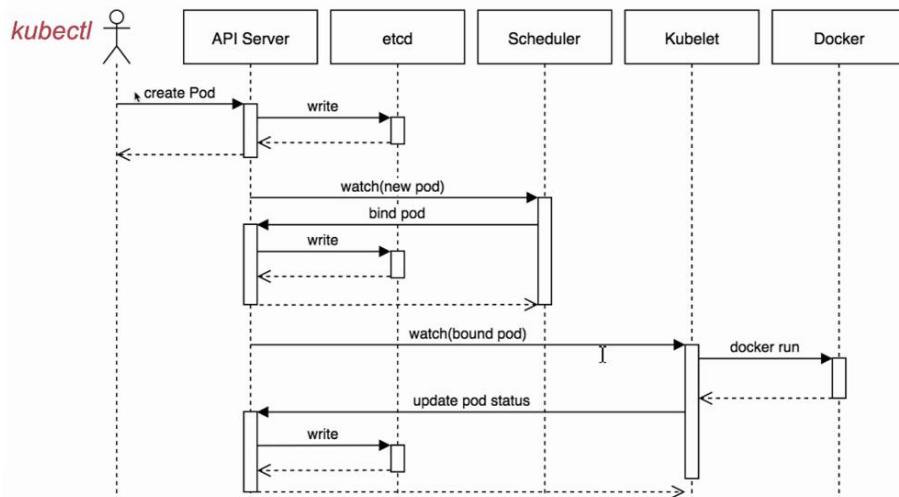
### ● kube-apiserver

- 暴露了一系列的REST接口, 连接client (如kubectl)和K8s集群的控制面
- 集群内部的应用也通过API Server来集群控制面交互
- 提供了身份认证、权限检查, 请求验证、访问控制等一系列的安全机制来保证控制面的安全



API Server 就是暴露了一些接口, 让用户可以和内部 K8S 做交互, 内部的组件也可以使用 API Server 暴露的 API。既然我们是一个守门人, 外面的人和里面的人都要找我们, 这会涉及到一些安全问题, 我们要阻挡一些恶意请求, 那么我们就要做一些认证和检查工作来保证我们系统的安全。

## 从kubectl到K8s集群内部 (3) : 完整流程



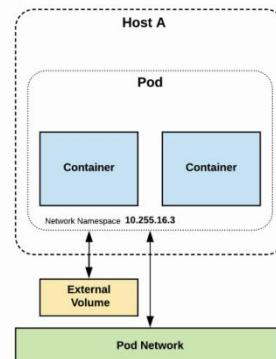
我们首先去 etcd (元数据管理、保证系统一致性)，我们去写一些元数据相关的东西。然后通过 watch 告诉 scheduler 告诉我们要做一个 new pod，然后把 bound pod 发给 kubelet，也就是 Pod 绑定到哪个节点上。然后 Kubelet 通过调用 docker run 真正部署这个结果。到 K8S 集群里，我们需要做很多个步骤才能完成启动 Pod 这个流程。

这些步骤之间可能还要考虑容错，需要一些容错的东西。我们用云原生的一些东西，之间的开销我们测出来会发现挺高的。

上节课已经介绍了 Pod 的结构，它是 K8S 管理的基础单元，也是我们 LAB 开始的基础内容。以前我们理解的是容器，现在我们是对容器包了一层做成 Pod 抽象。

## 回顾：Pod

- K8s 中最小的计算单元
- 和容器的区别
  - 包含多个容器
- 和容器的相同点
  - 共享网络：Pod Network
  - 共享存储：Volume (卷)



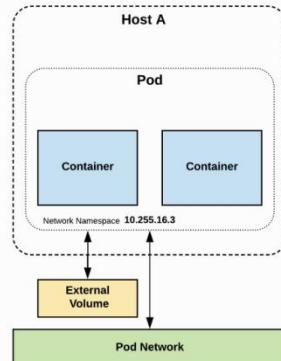
最大的区别就是包含多个容器，容器能做的事情 Pod 都能做，比如共享网络就是搞了一个虚拟网卡，存储可能就是配置了一些共享的数据卷。核心就是 Pod 是一个多容器，它有一个很重要的瞬时特性。在 Pod 模型里，把它的运行和存储是分开的，存储是在外部的。我们认为 Pod 内部没有存储的概念，都是挂载在外面或者通过网络连接。Pod 会因为各种各样的原因 crash。

## 回顾：Pod

- 很重要的一个特性：
  - Ephemeral (瞬息的)
- Pod 可能会因为各种原因挂掉



生产环境中通常使用更高抽象层次的一些 Objects，结合 Replication Controller 等机制，来保证服务的可用



36

crash 之后，我们就需要做一些处理，可以使用更高层次的一些 Object，比如 Replication Controller 去恢复一个 Pod，它是在 Pod 的基础上去描述它的数量和服务的可用性。

## Pod是API对象

- 和 API Server 对应的是 API 对象，基于 Rest (Representational state transfer) 接口
- K8s 中的所有资源都是 API 对象

听说 K8s 内部有大量的对象，怎么管理他们呢？



小明

kubectl 就是通过接口做 API 对象的管理，具体怎么管理呢？就是我们的 API 组的概念。

## API 组 (Groups)

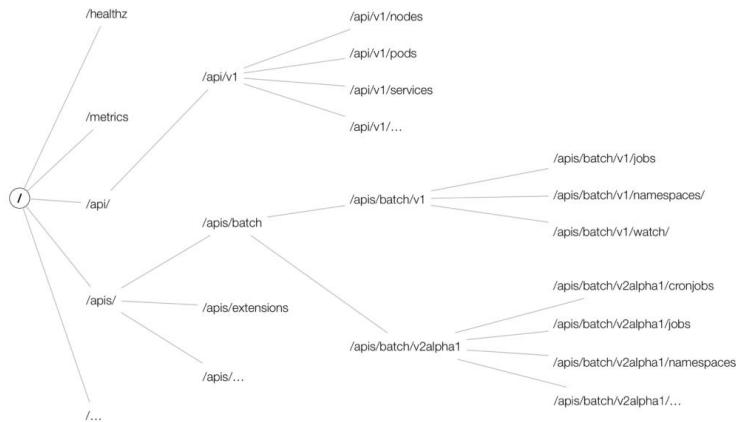
### API组 (Groups)

- API组:**
  - REST (接口) 路径
  - 每个 object 都属于一个 API 组
  - 每个 Object 的属性中的 `apiVersion` 和 `kind` 对应这里的 API 组中的信息

格式:  
`/apis/<group>/<version>/<resource>`  
例子:  
`/apis/apps/v1/deployments`  
`/apis/batch/v1beta1/cronjobs`

API 组整体也是一个很简单的组织，首先每个 API 对象属于一个 API 组。API 对象的属性其实对应 API 组里的一些信息。

API 组是拿斜杠作为划分，每个斜杠后都是 namespace。



所以我们最终可以形成这样树状的结构，从 API 根目录下面我们可以逐渐展开。通过这样的方式，我们通过 URL Path 的方式来管理这样的 API 对象。

这里的 `api` 对应的是没有名字的一个特殊的组，独立在 `apis` 以外的一个资源。而 `apis` 里可以继续展开，从而得到 `batch`、`extensions` 等的组。

API版本

- **K8s维护三个不同的API成熟度**
    - 有利于生产和开发环境的区分使用
  - **Alpha:**
    - 可能出现bug和变更，默认是关闭的
  - **Beta:**
    - 经过完整的测试，相对稳定，但是API的格式等可能会变化，默认开启
  - **Stable:**
    - 最终发不出来的稳定版本，接口不会变化，默认开启

/apis/<group>/<version>/<resource>

例子：

/apis/apps/v1/deployments

/apis/batch/v1beta1/cronjobs

Alpha 属于内测的状态，最终是 Stable 是发布状态的稳定版本。

## 对象模型 (Object Model)

- 对象 声明式
    - 描述集群内部的对应对象的期望状态
    - “Record of intent”
  - 所有对象必须包含:
    - apiVersion
    - kind
    - metadata.name, metadata.namespace, 和metadata.uid.
    - 对于workload类型的应用：还有spec和status

在 API 管理之下，就是一些对象了，比如上节课讲到的 `deployment` 对应的就是 `docker run` 真正执行的时候的对象。其实对象描述的也是对应对象的期望状态，核心还是以期望状态来作为 K8S 看到的衡量指标，只有知道这个，K8S 才能知道自己要怎么操作。

期望状态是声明式的管理方式，因为它没有具体描述我们的期望状态如何达到。

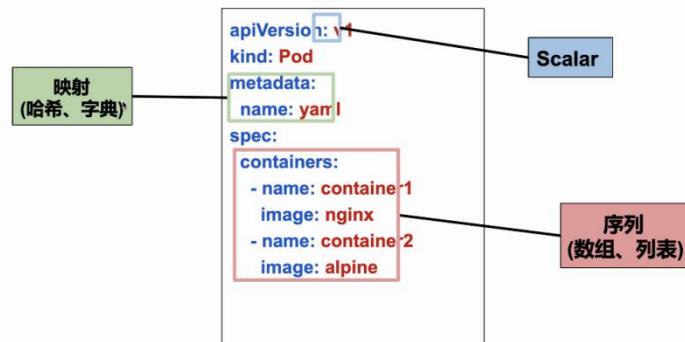
- **apiVersion**: 对象对应的K8s API版本
  - **kind**: k8s对象类型
  - **metadata.name**: 对象名
  - **metadata.namespace**: 对象所属的环境名 (不同的namespace可以理解为是不同的环境)
  - **metadata.uid**: k8s为对象生成的标识符(uid)

我们根据上图的例子来看。我们可以认为 namespace 是对象所属的环境名字，比如开发环境、调试环境、部署环境。uid 有点像 docker container 的唯一标识符。

## 对象表达：YAML

- 当使用文件等方式表达K8s对象时，通常使用YAML格式
- 一个“Human Friendly”的数据序列化格式
- 通过空格、对齐等方式来分层管理信息
  - 基本的三个数据类型：
    - 映射 (mappings): 哈希或字典的方式
    - 序列 (sequence): 数组或列表
    - Scalars: 字符串、数字、布尔值等

YAML 就是一种标记语言。它核心就是“Human Friendly”的格式。最早部署网站的时候，可能网站模板已经写好了一个 YAML 文件，这就要求内容是容易看懂的。这个格式的核心就是缩进。之前的 HTML, XML 都是使用标签的。



这边就是一个例子，以缩进的方式来表示我们的层级。-就是包含多个容器。

## 对象表达：YAML v.s. JSON



JSON 和 YAML 是完全可以对应的。

## Workload（负载）

- “A workload is *an application* running on Kubernetes.”
- Pod 是 workload 的核心对象
  - 然而由于 Pod 本身是瞬时的，不建议用户直接通过 Pod 来跑应用
- K8s 提供了一系列的 workload 对象，来帮助用户管理应用
  - Deployment 和 ReplicaSet (RC 的下一代)：通常用来管理无状态的应用程序
  - StatefulSet：允许需要有状态或者持久化数据的 Pod
  - DaemonSet：实现 node-local 的一些机制，通常用于插件等场景
  - Job & CronJob：用来管理 run-to-completion 类型的任务

workload 简单来说就是跑在 K8S 上的应用，任何服务、监控进程、调试进程，都可以认为是 workload（负载）。一个 workload 既然对应了应用，那么显然 Pod 是 workload 里很核心的概念。

Pod 更多描述的是运行时的状态（瞬时特性），当它挂掉之后，Pod 就不存在了。这样一个瞬时的东西，就导致我们容错、恢复、持久化很难使用 Pod 去实现，没有办法使用 Pod 去表示。

在 K8S 里，我们不用 Pod 跑应用，它是受 workload 对象的管理之下来跑应用的。这里把概念划分得比较细，Pod 衡量的是运行时的状态，可以认为是一个 stateless 的东西，所有持久化的东西就放在外部。我们之后学 serverless 也是类似的，更多刻画的是运行时的状态，这样我们是很容易做 auto-scale 和 replica 的。

比如我们起了一个网站和监控网站流量的监控软件。我们写了和 Pod 相关的策略，如果 CPU 水位过高，那么我们就扩容。这样 K8S 是不需要管 Pod 状态是什么样子的，K8S 不在乎此时 Pod 是否在操作数据库，因为 Pod 是瞬时的东西，所以 K8S 可以直接在有需要的时候扩容。我们扩容无非就是启动一个新网站和一个新的监控容器。因为 Pod 不管错误之后的状态，所以对错误的恢复可能有问题。

所以我们在 Pod 基础上实现 workload 来帮助用户管理应用，比如 deployment 可以做 Pod 自动的部署，部署可以有效帮助我们部署一个新的 Pod 和容器。ReplicaSet 就可以管理无状态的应用，一旦 Pod 挂了，我们就可以启动一个新的 Replica。

- K8s 提供了一系列的 workload 对象，来帮助用户管理应用
  - Deployment 和 ReplicaSet (RC 的下一代)：通常用来管理无状态的应用程序
  - StatefulSet：允许需要有状态或者持久化数据的 Pod
  - DaemonSet：实现 node-local 的一些机制，通常用于插件等场景
  - Job & CronJob：用来管理 run-to-completion 类型的任务

如果和数据库相关的，比如正在做事务处理，所以我们可能需要 StatefulSet 来考虑 Pod 有持久化数据的情况下该怎么做扩容。DaemonSet 就是可以做一些插件。

什么是 run-to-completion 呢？比如线程 T1 和 T2 做上下文切换的时候，是 OS 决定的。

如果 OS 不知道 T1 和 T2 的关系，可能就会调度出一个很差的结果。我们就可以做用户态的协程，T1 做完通过 yield 直接切换到 T2 做。这样我们就可以做一些 run-to-completion 的任务，可以做一些任务的组合。

- 和Workload(负载)相关的Object，有两个重要的属性

### spec and status

- spec – 描述一个对象的期望状态 (或者是描述对象的配置)
- status – 由K8s来维护，用来描述一个对象的真实状态以及相关的历史信息 (比如信息的变更历史)

workload 相比 API 对象还加了两个新的属性，一个是期望的状态，还有一个是真实的状态。

例子：spec 和 status

#### 案例：对象

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
    - name: nginx
      image: nginx:stable-alpine
      ports:
        - containerPort: 80
```

#### 案例：部分Status内容

```
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:52Z
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:49Z
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: 2018-02-14T14:15:49Z
    status: "True"
    type: PodScheduled
```

比如这里容器是 nginx，有一个 80 的端口，这里没有什么定量的东西，这也属于期望状态。在这样的状态下，status 就是看一看状态是否满足了我们的 spec 的要求，比如端口是否是 80。我们看到 status 和 spec 不一定是完全对应的。

spec 描述了容器的配置，而 status 里描述的更像是日志和历史信息，打印出来的东西不一定完全是 spec 里描述的信息。status 可以认为是运行时展示出来的状态的更大的一个集合，而 spec 是运行前开发者对期望运行状态的一个描述。

## K8s 的核心对象

我们已经讲了 Pod, Service 等核心对象。

- K8s内部虽然有复杂的结构和对象
- 不过这些对象基本上是基于底下的核心对象及其抽象构建的
  - Namespace
  - Pods
  - Labels
  - Selectors
  - Services

主要有命名空间、Label、Selector、Services。Label 和 Selector 帮助我们对对象进行分类，对于 Pod 的属性和对属性的选择。

# 命名空间 (Namespaces)

## 命名空间 (Namespaces)

- 命名空间是一个逻辑上的集群或者独立环境
- 主要用来划分一个集群内部的资源，限定区域

```
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    app: MyBigWebApp
```

```
$ kubectl get ns --show-labels
NAME      STATUS  AGE   LABELS
default   Active  11h   <none>
kube-public Active  11h   <none>
kube-system Active  11h   <none>
prod      Active  6s    app=MyBigWebApp
```

第一个是 Namespace，它描述的可能更像是外部的环境。把我们一个系统划分成不同的部分。这里有一个例子，我们的 K8s 里面有一些默认的 Namespace。prod 就是一个 production namespace，就是在生产环境中使用的命名空间。注意到 ns 也是一个核心对象，我们可以通过 kubectl 来 get ns 对象。

## 默认的一些命名空间

- Default:**
  - 所有的对象默认的命名空间
- kube-system:**
  - 存放着所有K8s自己创建的对象或资源 (类似系统资源)
- kube-public:**
  - 特殊的命名空间，可以被所有的用户所访问
  - 主要用来做cluster bootstrap和配置相关的任务

```
$ kubectl get ns --show-labels
NAME      STATUS  AGE   LABELS
default*  Active  11h   <none>
kube-public Active  11h   <none>
kube-system Active  11h   <none>
```

前面的这三个主要是 K8s 默认创建的命名空间。如果大家不做命名空间分配的话，对象都会去 Default 空间。Public 比较特殊，它可以被所有用户访问，可以做 cluster 启动和配置相关的任务，怎么使用是和用户相关的。我们可以把共用的服务和对象放到 public 命名空间里去。

我们回顾一下前面的例子：

Pod 的对象中，包含 namespace 的信息表明，这个 Pod 是处在 Default 这个命名空间中的

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  namespace: default
  uid: f8798d82-11e5-11e8-94ce-080027b3c7a6
```

这个 Pod 的元数据里就描述了它的名字和命名空间，我们没有对它进行分类就放在默认的空间中。这可能并不是一个好的方法，还是可以通过 kubectl 来创建新的 namespace。

有了命名空间之后，用户就可以自己去管控自己的资源。

## Pod

### Pod

- 大家已经比较熟悉的一个对象



复杂例子中我们 mount 了更多的数据卷，还传入了参数。在我们使用 docker 的时候，可以配置 docker command 是什么，这些所有的配置最终也是变成我们的 container 里的一些内容。

### Pod Container的关键属性

- name** – 容器名
- image** – 容器镜像
- ports** – 一组需要暴露的端口，同样可以指定对应的协议等
- env** – 一组环境变量 (environment variables)
- command** - 入口点(和Docker ENTRYPOINT 等价)
- args** – entry对应的命令的参数 (等价于 Docker CMD)



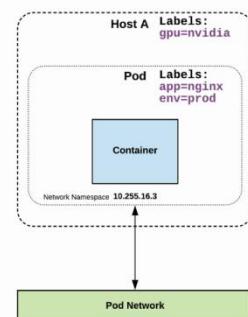
最终 kubectl 也是调用相应的 docker 参数来做启动。这都可以和我们的 docker 内容映射起来。

## Labels

### Labels

- 键值对：用来标识各种资源他们的关系
- 不能使用Label来做资源的唯一标识符！**
- Label的名称有特定的格式要求

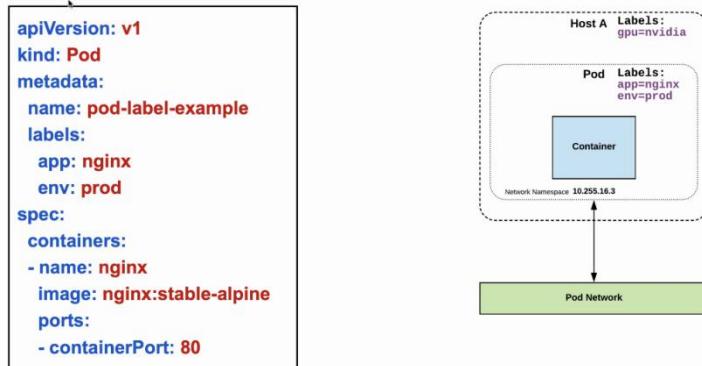
\*



Label 其实就是顾名思义，对我们的对象打一些标签来标识我们资源的关系。比如这里

我们就是给 Host A 来打一个标签。比如配置 Host A 的 gpu=nvidia。我们在 Pod 里也可以配置自己的标签，如 app=nginx。这些标签描述了我们的属性。但是我们不能使用 label 作为资源的唯一标识符。

## Labels使用案例



我们刚才的案例中，在我们配置一个 Pod 的时候，就可以在里面加 label。

## 选择器：Selectors

### 选择器 : Selectors

- Selector可以通过过滤 label，来选取一组特定的对象，在K8s内部中广泛使用着

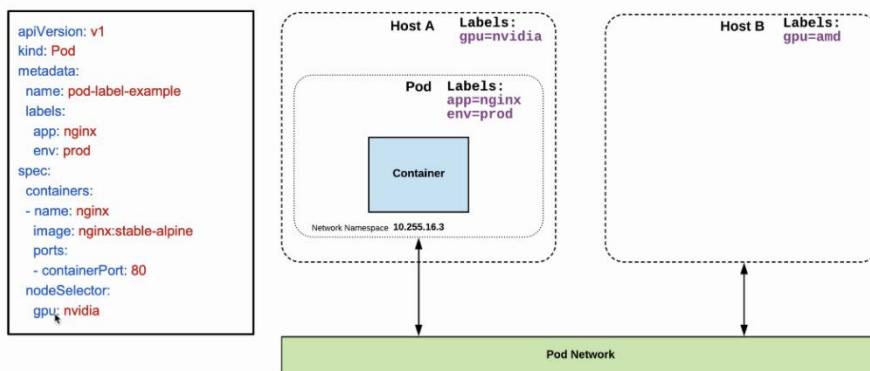
```

apiVersion: v1
kind: Pod
metadata:
  name: pod-label-example
labels:
  app: nginx
  env: prod
spec:
  containers:
    - name: nginx
      image: nginx:stable-alpine
      ports:
        - containerPort: 80
  nodeSelector:
    gpu: nvidia

```

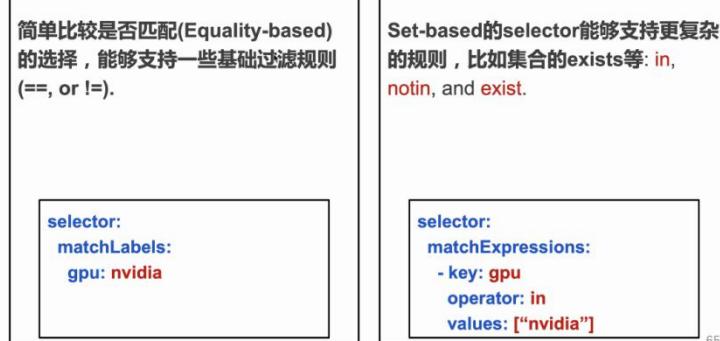
Pod 不是要部署到不同的节点上吗，我们就可以通过配置 nodeSelector 来考虑我们的期望的需求，比如我们期望 gpu 是 nvidia 的。K8S 就是通过这个 selector 在节点中进行选取。

## Selectors使用案例



在这种情况下，我们就达成了通过 selector 实现节点的选取。

## Selector 类型

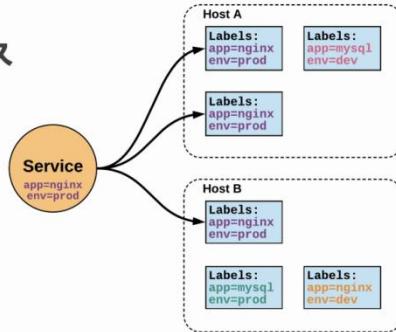


随着 K8S 的发展，我们的 selector 也越来越强大。比如可以设置 expression 等复杂规则。在 YAML 下因为对格式有限制，所以可能需要写出这样的组合形式。

## Service

### Service

- 用来封装Pod
- 相比Pod，Service是持久(Durable)的K8s资源
  - 静态cluster IP
  - 静态DNS name (在某个namespac内)



Service 在 Pod 基础上建立了持久化的资源，并且会考虑到 crash 之后我们的资源怎么处理。我们可以看到 Labels 里有 app 和 env。Service 可以包含静态的 IP，也可以有静态的 DNS 名字。这里我们简单地说了一些 Service，下节课会主要讲 Service，再之后会讲 workload 对象等。

## 下节课：Kubernetes (三)

- K8s的5个核心对象(最后一个): Service
- K8s中的workload对象: ReplicaSet、Deployment、DaemonSet、StatefulSet、Job、CronJob
- Pod的网络(k8s网络模型)
- k8s存储(Volume)
- 从k8s到minik8s

## Reference for Lab (1)

- Kubectl: <https://kubectl.docs.kubernetes.io/guides/>

2022/4/15

上节课我们提到 K8s 是基于声明式的管理，上节课我们也讨论了声明式（去餐馆点单）和命令式（指令明确，每一步规定了）的区别。

## 回顾：命令式与声明式



3

在我们 kubectl 的时候，有一些命令式的使用方法，会相对来说比较 low-level，比如 create 做的行为就是比较具体的。

## 回顾 : kubectl

- K8s提供的命令行工具，允许用户/开发者和一个K8s集群的控制面(Master)进行交互



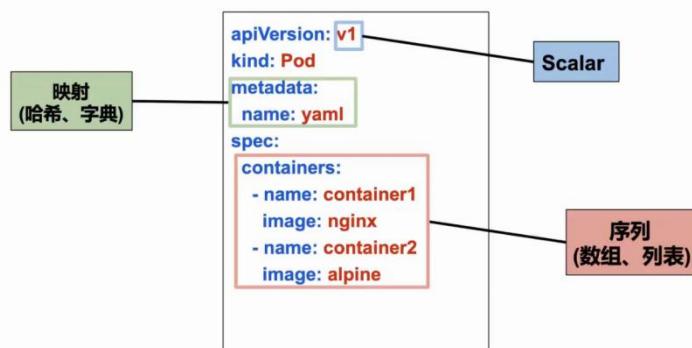
## 回顾 : K8s对象模型

- `apiVersion`: 对象对应的K8s API版本
- `kind`: k8s对象类型
- `metadata.name`: 对象名
- `metadata.namespace`: 对象所属的环境名 (不同的 namespace可以理解为是不同的环境 )
- `metadata.uid`: k8s为对象生成的标识符(uid)

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
  namespace: default
  uid: f8798d82-11e8-94ce-080027b3c7a6
```

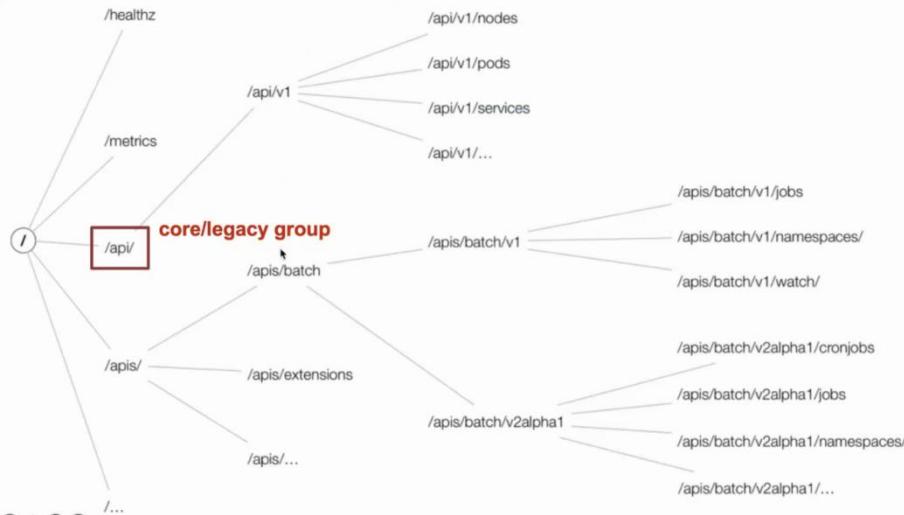
K8s 里万物皆对象，尽可能把所有对外展现的 API 都抽象出对象。对象的字段包括了：版本、类型、元数据。`namespace` 是另一种环境对象，可以把一些 Pod 分开。实际上这个结构是基于 YAML，它是靠缩进进行管理的格式。

## 回顾 : K8s对象表达 (基于YAML)



YAML 支持的类型包括标量、序列等。

## ▶ 回顾：API组 (Groups)



API 组是按照像文件路径这样的格式去划分的。而 /api/ 是叫做 core/legacy group 的组。

## 回顾：K8s的5个核心对象

- **Namespace:** 命名空间是一个逻辑上的集群或者独立环境
- **Pods:** 基本计算单元（容器属性）
- **Labels:** 表示K8s内部的资源或对象的关系
- **Selectors:** 基于Label来选取一组对象
- **Services:** ?

我们衍生出了五种核心的类型，再次基础上就可以开发复杂的对象。Pod 里可以包含多个容器。Label 和 Selector 关系比较近，Selector 可以基于 Label 来选取对象。

## Services

它的名字是服务，就是持久化的资源。

### Services

- 访问Pod中运行的任务/应用暴露的服务的方法
- 持久化的 (Durable) 资源
  - Pod是非持久化的，随时可能挂掉
  - static cluster-unique IP
  - static namespaced DNS name

**<service name>. <namespace>. svc.cluster.local**

大家可能认为 Pod 和 Service 的区别只是在于有没有持久化的数据，这个理解不是很准确。Pod 可以通过挂载数据卷的方式来使用持久化的数据。Pod 和 Service 都可以使用持久化

的资源。

服务就是 24 小时不间断的服务，比如水龙头。在定义服务的时候，K8s 就设定了如果 Service 挂掉的话，是能够自动重启的。这是 Pod 和 Service 最主要的不同。Service 认为是需要长期提供服务的。

因为 Pod 不关心挂了以后会怎么样，希望应用自己有能力去处理。Service 为了有能力重启，K8s 内部需要做一些功能。

比如 K8s 需要给每个 Service 给一个静态 IP，也就是说重启以后还是使用这个静态 IP。这样对其他应用来说，还是可以使用原先的 IP 来发现重启的服务。还有给一个固定的 DNS 的名字，只要我们走 DNS Lookup 能够解析到绑定的服务器，那也可以。如下是 K8s 中对服务的域名的规定：

<service name>.<namespace>.svc.cluster.local

在这样情况下，我们每一个服务的域名就被固定下来了。重启之后，我们还是可以把 DNS 域名重新给到这个 Pod，重新提供一个服务。这样我们就可以让 Service 提供一个长时间的持久服务。

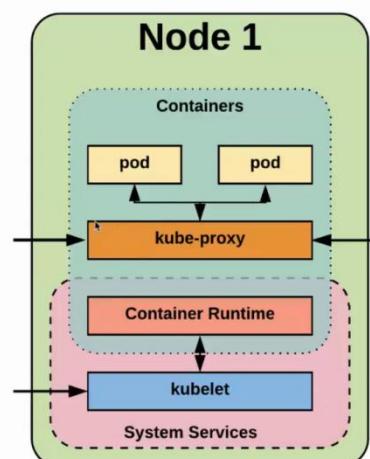
## Services (2)

- 通过 Equality-based Selector 来定位目标 Pod
- 通过 kube-proxy 来提供简单的负载均衡
- 回顾：
  - Kube-proxy 是一个 Per-Node 的后台服务
  - 会在 Node 的 iptable 中为每个 Service 创建对应的条目

Selector 有两种，一种是 Equality-based，另一种是 Set-based。在定位之后，我们可以做简单的负载均衡。网络是 K8s 中比较复杂的部分，当我们去部署服务的时候，有 IP、DNS 等。在容器和 K8s 里面，可能要实现和网络相关的服务，比如把服务映射到 DNS 名字，把 DNS 名字映射成 IP。

## Kube-proxy 回顾

- Node 上的组件
  - kubelet
  - **kube-proxy**
  - Container Runtime Engine



Kube-proxy 其实是每个节点里的一个组件，proxy 的核心功能就是提供后台服务。能让

不同的节点之间相连。比如对一个 IP 的请求过来了，`kube-proxy` 需要知道转发给这个节点中的哪个 Pod。Kube-proxy 其实就是在 `iptables` 里为每个 Service 创建条目。这样我们就可以去管理 Service，Proxy 知道哪个 ip 对应哪个 Pod。

## Kube-proxy回顾(2)

- 管理每个Node上的网络
- 负责连接的转发以及K8s Service的负载均衡等任
- 支持的Proxy模式：
  - Userspace
  - iptables
  - Ipvs等

Kube-proxy 其实除了负责转发以外，它还负责一些负载均衡的任务，这主要是在 `kube-proxy` 之间去做的。比如我们的 Pod 是一些相同的 replica，那么我们的 proxy 是可以做负载均衡的。proxy 的模式我们就不仔细去讲了，不同的 IP 的配置方法不一样。

# Service 类型

- 主要有四种常见的Service
  - ClusterIP (**default**)
  - NodePort
  - LoadBalancer
  - ExternalName

Service 主要分四种，大家可以看到这个类型是按照从里到外的类型去介绍的。最前面的最简单，越到后面越 high-level，service 可见的范围也越大。

我们先讲 ClusterIP Service，这是最简单的一种。它其实是通过虚拟 IP。因为我们现在是在 K8s 内部做网络和服务的管理，这些网络都是由 K8s 虚拟出来的。它的 IP 结构其实和我们平时见到的 IP 没什么区别，但是它都是内部管理的，主要由 K8s 的软件做的网络包转发、路由表等功能。

## ClusterIP Service

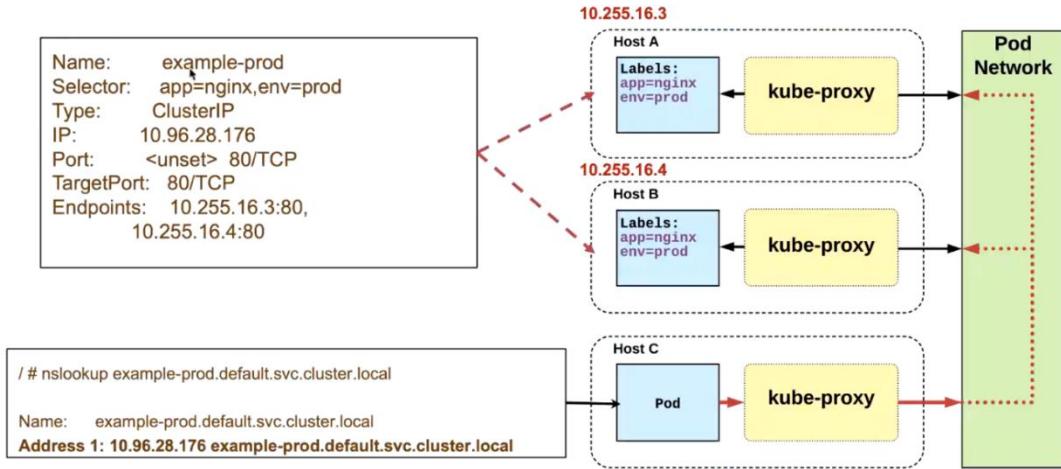
- ClusterIP services 通过使用集群内部的虚拟 IP (cluster-internal virtual IP) 来对外暴露服务

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  selector:
    app: nginx
    env: prod
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

其实大家看右边的 YAML 配置文件，这里面并没有和 IP 相关的东西。我们的种类是 Service，元数据和 spec 都和其它的 Object 差不多，无非是定义了协议和端口。

Q: YAML 里面并没有描述 ClusterIP，这是为什么呢？

A: 这是因为 ClusterIP Service 是最简单的一种服务，这个服务不需要标注 IP。只要我们这样写了之后，K8s 就默认是 ClusterIP Service 类型。



K8s 会把 YAML 转化成上图左边的配置，其中没有办法直接对应到 YAML 的是 IP, Endpoints。那么 K8s 是怎么做转换的呢？

首先 K8s 会自动识别它是一个 ClusterIP Service 类型，那么就需要分配一个虚拟 IP 给它。K8s 接下来会根据 Selector 中的需求来找到符合我们 Selector 的 Pod，比如在上例中我们找到了 Host A 和 Host B。我们找到这两个 Pod 对应的 IP 为 10.255.16.3 和 10.255.16.4 写在 endpoints 这里，也就我们的这个应用可以部署在这两个 Pod 上。但是我们还是需要给 Service 提供一个静态 IP，因为如果我们只根据 Host 的 IP 来进行索引的话，10.255.16.3 的 Host 挂了的情况下，它可能是硬件坏了，一时半会不能启动了。这时候我们可能需要 10.255.16.4 的 Host 去跑这个应用，所以我们最好使用 Service 的静态 IP 去找。我们的 Service 对用户是抽象的一个软件服务，它不应该依赖于硬件的环境的好坏，所以提供给用户的 IP 应当是稳定的。所以我们需要给 Service 一个静态 IP: 10.96.28.176。这样就构成了一个运行的环境。

对 Host C 来说，它不符合 Selector 的条件，所以我们不能部署到 Host C 上。

nslookup 是 DNS 域名服务发现的方法，我们可以直接通过这个 Service 的域名找到对应

的 Service 的静态 IP。之后我们就可以通过这个 IP 来访问这个服务。

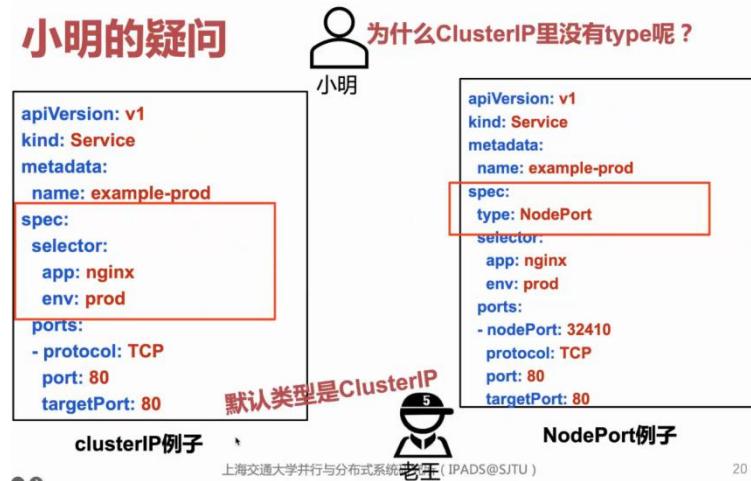
## NodePort Service

### NodePort Service

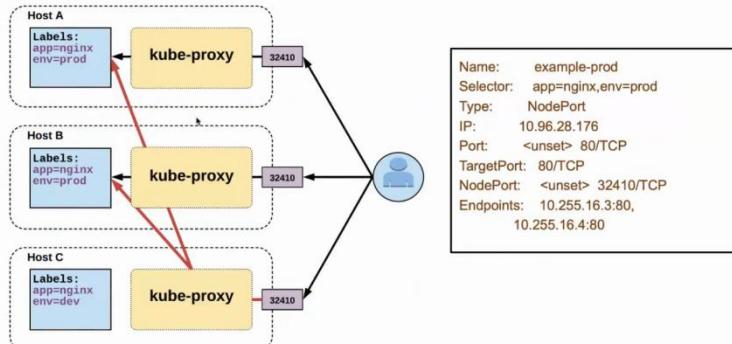
- NodePort services 扩展了 ClusterIP service
- 在每个Node的IP上，都暴露了一个端口，来访问对应的Service服务
- 这里的端口可以是静态指定的，也可以动态分配(通常区间为：30000-32767)

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: NodePort
  selector:
    app: nginx
    env: prod
  ports:
    - nodePort: 32410
      protocol: TCP
      port: 80
      targetPort: 80
```

我们可以看到这里就不一样了，在这里 spec 里我们就需要指定 NodePort。如果我们每个都使用 IP 去管理，还是比较麻烦的。我们比起记录 IP，不如记录端口。比如设置端口为 32410。



### NodePort Service



原先是通过名字找到 IP 找到服务，而现在 NodePort Service 是用端口的方式访问不同的服务的。我们可以看到右边的运行时信息，我们看到 IP 是没有变的，唯一的差别就是多了

一个 NodePort。K8s 实现成了一个很有趣的效果，它对于所有 namespace 里的节点全部部署成，只要访问端口 32410 就会访问到对应的服务上。哪怕我们的 Host C 并没有部署对应的服务，访问 Host C 的 32410 端口也会转发到拥有这个服务的 Host A 和 Host B 上去。我们公认在这个集群中，不管使用哪个 IP 访问 32410 端口，都会访问到这个 Service 上去。但是这个依赖于 kube-proxy 的配置。

这样我们就不需要去做 nslookup 了。我们只要访问自己机器的这个端口就能找到这个服务，好像我们的所有服务都跑在这个机器上一样。问题就是端口数量有限，可能没有 IP 方法可扩展性更好。

这里还可以注意的一点就是 kube-proxy 转发功能不太一样了。以前是应用部署了以后，proxy 可以在 Node 里做 load-balance。而现在的所有 proxy 都要转发来参与做负载均衡。

## LoadBalancer Service

能不能把这个特点提取出来呢，就是 LoadBalancer Service。

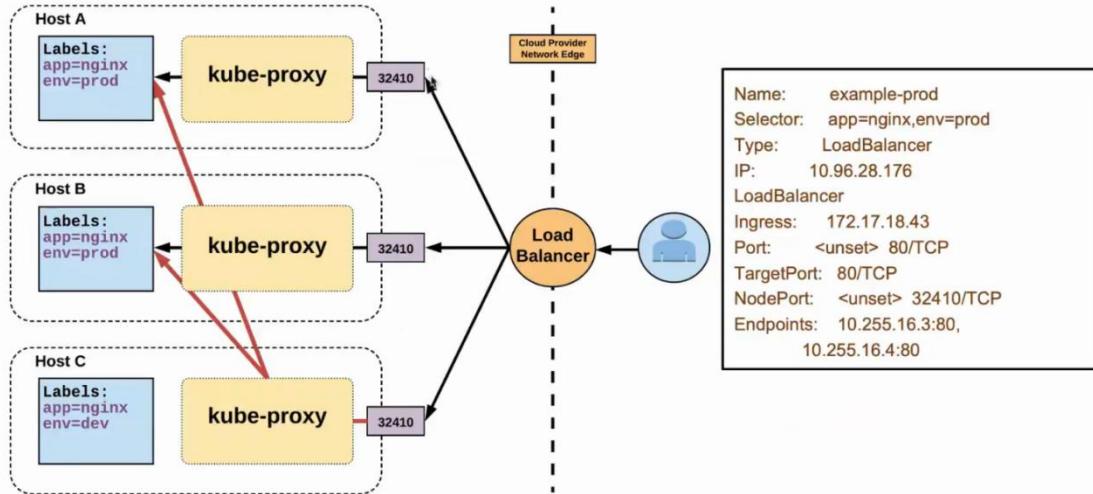
## LoadBalancer Service

- LoadBalancer service 进一步扩展了NodePort
- 将一个集群外部IP (cluster external IP)绑定到对应的 Service
- 通常需要结合某些外部系统来提供Ingress

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: LoadBalancer
  selector:
    app: nginx
    env: prod
  ports:
    protocol: TCP
    port: 80
    targetPort: 80
```

它把集群外部 IP 来绑定到对应的服务。LoadBalancer 这里也只需要设置一下 type 就行了。

# LoadBalancer Service



在实现的时候，我们会在服务的外部设置这个 Load Balancer。之前的 IP 划分都是 10. 开始，这很明显是内部虚拟 IP。Ingress 可能就是一个外部的 IP，这个 IP 可能就绑定到了我们数据中心环境下的每一台机器的对外 IP。服务器外部的 IP 之间是可以互相连接的。我们就把内部的服务绑定到了一个外部的 IP 上，这样以后我们就可以通过 172.17.18.43 进入。如果 K8s 分布式管理很多 Pod 和 Service，那么可能需要外部的 Balancer 来进行负载均衡是比较合理的。

## ExternalName Service

- ExternalName主要是用来索引集群外部的相关资源
  - 通过DNS域名等方式

```
apiVersion: v1
kind: Service
metadata:
  name: example-prod
spec:
  type: ExternalName
  spec:
    externalName: example.com
```

这个就是用来索引集群外部的相关资源的。我们直接使用 global DNS，来进行服务的发现。我们用这种域名来解析我们的 externalName。

## K8s Service小节

- 从ClusterIP到NodePort到LB
  - 抽象层次越来越高
  - 可能会引入更多跳，比如NodePort如果在没有对应的Endpoint的Node上执行，依赖于Kube-proxy进行额外的中转

在 IP 上我们抽象出端口，我们再引入一个外部 IP 来处理负载均衡，引入的层次和网络拓扑都越来越复杂。要访问外部服务就使用 ExternalName Service，使得 K8s 中的能力更强。

简单来讲，Service 就是为用户提供了长时间的服务。光讲这个东西，大家可能觉得实现还比较遥远，近期可能会出一些作业来大家熟悉一下。

## workload 对象

之前我们也提过 Workload 对象里 spec 和 status 可能会比较明显，因为负载实际上就是真正用来描述某一个应用场景的对象。这个对象不再是我们之前讲到的核心对象了，它的层次更高，包含的用户语义更多。

## Workloads

- Workload系列的对象是K8s中抽象层次较高的对象
- 通常用来管理Pods，或者基于Pod的高层次对象
- 在所有情况下，Workload系列的对象都需要包括一个Pod模板，来管理最基础的计算单元
- 课程中会介绍的Workload对象
  - ReplicaSet、Deployment、DaemonSet、StatefulSet、Job、CronJob

从 Pod 模板生成 Workload 可能是一个自动的过程。

## Pod模板

- Workload的控制器(Controllers) 基于用户提供的Pod模板来管理Pod的实例
- Pod模板：Pod的对象Spec简化版本 (包含更少的metadata项)
- 控制器会通过Pod模板来创建真正的运行的Pod实例

Pod Spec	Pod 模板
<pre>apiVersion: v1 kind: Pod metadata:   name: pod-example   labels:     app: nginx spec:   containers:     - name: nginx       image: nginx</pre>	<pre>template:   metadata:     labels:       app: nginx   spec:     containers:       - name: nginx         image: nginx</pre>

就是在 spec 上做了简化。在 workload 创建的时候，有一些条目我们可能不太关心，比如 kind, apiVersion 我们都不关心。

## ReplicaSet

我们来看第三个要做的功能：ReplicaSet，我们之前已经介绍过 ReplicationController (RC) 了。也就是我们可以设定期望的 replica 数量，一旦和期望数量不符了，我们就可以自动创建 replica。后来 K8s 就统一为 ReplicaSet 了。

- 前面介绍过的ReplicationController(RC)的升级版
- 非常常用的管理Pod备份实例和生命周期的对象
  - 管理Pod的调度、扩容(scaling)、删除等
- 核心任务：始终保证期望数量(Desired)的Pod实例在运行



ReplicaSet 也不一定要用于容错，比如我们的服务过载了、不够用了，我们就可以加新的 replica，通过负载均衡器去分派到不同的 Replica 上。这个主要是为了降低过载的问题，降低延迟来提升用户体验。

## ReplicaSet

- **replicas:** Pod的期望实例数
- **selector:** Label selector用来管理该ReplicaSet的目标Pod
  - 例子，app为nginx，env为prod(生产环境)的Pod是该ReplicaSet的目标Pod

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  template:
    <pod template>
```

RC 可能是在 master 上运行的控制器，而 ReplicaSet 可以抽象成专门的服务或对象，包含相关的内容。replica 字段就是 Pod 期望的实例数，而 selector 就要去管理 ReplicaSet 的目标 Pod。同一个 Pod 内，如果 Label 不一样，replica 不一样，我们也会认为它是两个 Pod，比如测试环境中的 Pod 和生产环境中的 Pod。这样我们可以把环境和 Pod 分开。

## ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: rs-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  template:
    metadata:
      labels:
        app: nginx
        env: prod
    spec:
      containers:
        - name: nginx
          image: nginx:stable-alpine
          ports:
            - containerPort: 80
```

① Label转换为对应的 selector  
② 三个运行实例

③ name: replicaset名+5字符的随机字符串

```
$ kubectl get pods 随机字符串
NAME READY STATUS RESTARTS AGE
rs-example-9l4dt 1/1 Running 0 1h
rs-example-b7bcg 1/1 Running 0 1h
rs-example-mkll2 1/1 Running 0 1h
```

```
$ kubectl describe rs rs-example
Name: rs-example
Namespace: default
Selector: app=nginx,env=prod
Labels: app=nginx
        env=prod
Annotations: <none>
Replicas: 3 current / 3 desired
Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels: app=nginx
          env=prod
  Containers:
    nginx:
      Image: nginx:stable-alpine
      Port: 80/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
  Events:
    Type Reason Age From Message
    ---- ---- -- -- -----
    Normal SuccessfulCreate 16s replicaset-controller Created pod: rs-example-mkll2
    Normal SuccessfulCreate 16s replicaset-controller Created pod: rs-example-b7bcg
    Normal SuccessfulCreate 16s replicaset-controller Created pod: rs-example-9l4dt
```

这边展示了一个 ReplicaSet 的例子，左边是 ReplicaSet 的描述，spec 还是在讲容器是什么样子。如果我们使用 kubectl get pods 的话，因为我们规定了 replica 的数量是 3，所以它会生成这三个 replica。describe 命令还是挺有用的，可以去看启动以后这个命令长什么样子。

## Deployment

在这个基础上，我们实现了 Deployment

- 在ReplicaSet的基础上，进一步封装的workload对象
- 通过声明式(Declarative)的方式来管理Pod
- 提供了回滚(rollback)机制和版本升级控制
- 版本升级控制通过: *pod-template-hash* 的标签控制
- 每一次升级迭代，会给对应的ReplicaSet和Pod打上新的标签



我们上节课讲 `docker create` 这样的指令的时候，我们是拿 `deployment` 抽象和一些功能进行对应的。我们可以认为 `docker run` 这样的功能就是在进行一些实际的部署。当然它也是声明式的方法来进行管理的。从 `ReplicaSet` 开始我们都是写 `spec` 来描述，都是声明式的方法了。

`Deployment` 主要提供了回滚（版本降级）和版本升级的机制。我们这节课主要讲版本升级控制。选微服务的同学要注意一下，版本升级是微服务要求的功能。

## Deployment

- **revisionHistoryLimit:** Deployment需要保留的历史迭代版本的数量
- **strategy:** 描述了Deployment中，升级Pod所使用的策略，主要有两个合法选项 **Recreate** 和 **RollingUpdate**.
  - **Recreate:** 在新版本的Pod启动前，需要先销毁所有的旧版本Pod
  - **RollingUpdate:** 滚动更新，根据两个参数：`maxSurge` 和 `maxUnavailable`，进行滚动更新

```

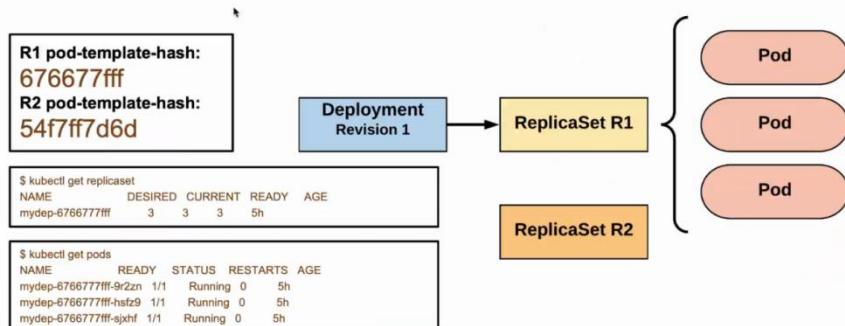
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deploy-example
spec:
  replicas: 3
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
      env: prod
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    <pod template>
  
```

更新时，维护的Pod实例数大概在: `[replicas-maxUnavailable, replicas+maxSurge]`

超过这个数量我们就不保留了。版本主要还是指的是 Pod Template，保留在运行的内存里。Recreate 全部关了 Pod 再升级，会涉及到服务暂停。还有一种是滚动升级。

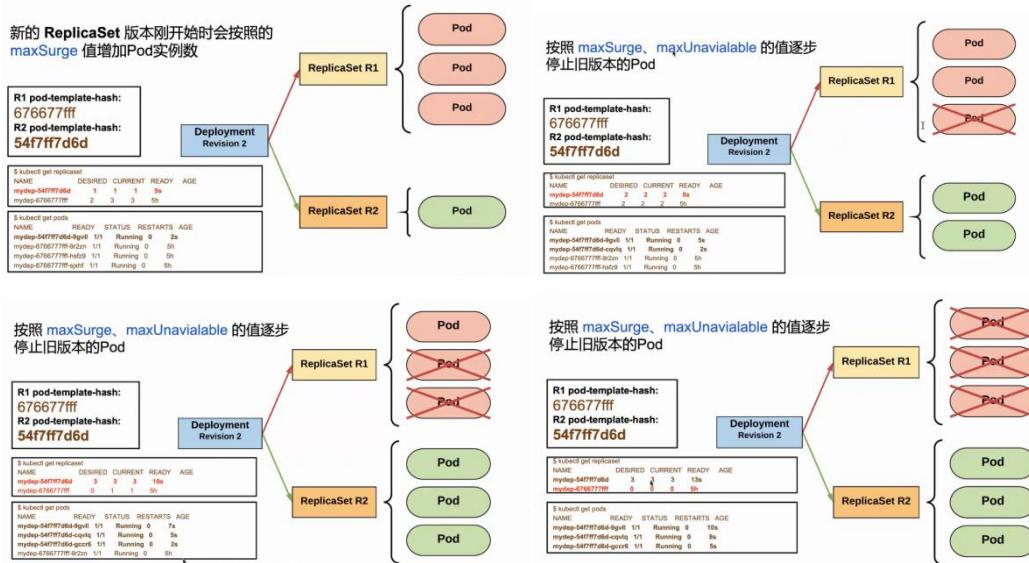
## Deployment与滚动更新 (或滚动升级)

更新Pod模板会生成一个新的 `ReplicaSet` 版本 (revision)



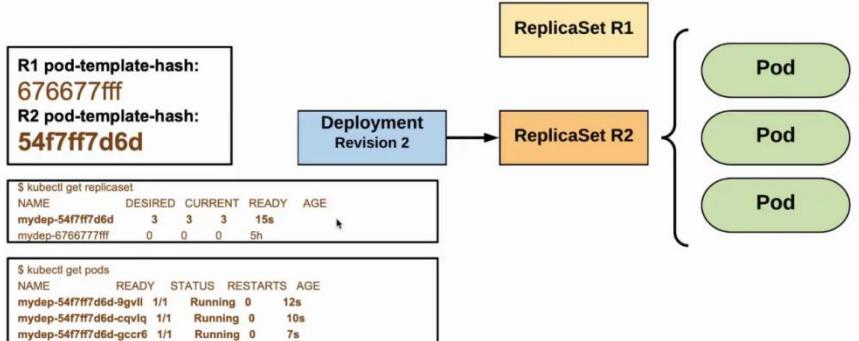
现在我们更新 Pod 模板就会生成一个新的 ReplicaSet R2 版本。此时我们还没有更新，属于准备阶段。这时候 kubectl 看 replicaset 的话，我们只发现了 R1。

接下来，我们就开始滚动升级。第一步增加一个 R2 的 Pod，此时我们多了一个 Replica Set 刚刚启动，有一个 Pod 是新的。



这样我们就完成了滚动升级。因为我们之前配置了 revisionHistoryLimit，此时我们还是能在 ReplicaSet 看到原来的 R1 的。在滚动升级的时候，维持了 Pod 实例数的大致的稳定，因为如果 Pod 少了，用户就能明显地感受到升级，体验就比较差。

### 滚动升级完成



## 滚动更新的好处

- Deployment相比ReplicaSet的主要特点
- 能够实现较为缓和的版本升级
  - 如果新版本出现问题，仍然有旧版本的Pod在运行和处理请求
  - 没有明显的Down-time时间
- 后面会介绍的其他Workload都支持了版本的升级管理

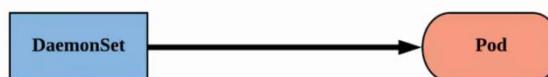
Q: 滚动更新有那么多好处，什么情况下必须要用 recreate（关掉全部旧版本再更新）的方法呢？

A: 旧版本有问题的情况，比如旧版本有严重安全漏洞。否则旧版本的 Pod 就会被人攻击。

## DaemonSet

这个和 ReplicaSet 比较像，更多是由 K8s 系统去使用，不太给应用使用。

- DaemonSet 会保证所有满足要求的 Node 上都会运行一个对应的 Pod 实例
- 这些 Pod 实例会绕开默认的调度机制
- DaemonSet 通常用来实现一些集群层面的服务，比如日志转发，健康状态检测(health monitoring)等
- 版本升级通过 *controller-revision-hash* 标签来管理和维护的



## DaemonSet

- **revisionHistoryLimit:** DaemonSet 需要保留的历史迭代版本的数量
- **updateStrategy:** 描述了 DaemonSet 中，升级 Pod 所使用的策略，主要有两个合法选项 **RollingUpdate** or **onDelete**.
  - **RollingUpdate:** 根据参数 `maxUnavailable` 进行滚动升级
  - **onDelete:** 新版本的 Pod 实例只能在现有的实例被删除后才能被部署上去

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ds-example
spec:
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    spec:
      nodeSelector:
       .nodeType: edge
<pod template>
```

只有上限，没有下限，最多删除的数量可能是 1.

## DaemonSet

- **spec.template.spec.nodeSelector:**
  - 用来确定目标的Node的Selector
- **DaemonSet会在所有满足Selector要求的Node上运行对应的Pod**

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ds-example
spec:
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
  updateStrategy:
    type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
  template:
    spec:
      nodeSelector:
       .nodeType: edge
<pod template>
```

这里的 template 里还多了一个 NodeType，用来确定在哪些节点上跑我们的 deamon。我们是在所有满足要求的 Node 上运行，所以我们不需要配置 replica。下面我们只找到了一个符合要求的 Node，所以就只配置了一个 Pod。

## DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ds-example
spec:
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: nginx
  updateStrategy:
    type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      nodeSelector:
       .nodeType: edge
      containers:
        - name: nginx
          image: nginx:stable-alpine
          ports:
            - containerPort: 80
```

```
$ kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
ds-example-x8kkz  1/1     Running   0          1m
```

```
$ kubectl describe ds ds-example
Name:           ds-example
Namespace:      default
Selector:       app=nginx,env=prod
Node-Selector: .nodeType=edge
Labels:         app=nginx
                env=prod
Annotations:   <none>
Desired Number of Nodes Scheduled: 1
Current Number of Nodes Scheduled: 1
Number of Nodes Scheduled with Up-to-date Pods: 1
Number of Nodes Scheduled with Available Pods: 1
Number of Nodes Misscheduled: 0
Pods Status:   1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=nginx
           env=prod
  Containers:
    nginx:
      Image:  nginx:stable-alpine
      Port:   80/TCP
      Environment:  <none>
      Mounts:  <none>
      Volumes: <none>
  Events:
    Type  Reason  Age   From           Message
    --  --  --  --  --
    Normal  SuccessfulCreate  48s  daemonset-controller  Created pod: ds-example-x8kkz
```

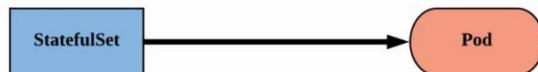
上海交通大学并行与分布式系统研究所 ( IPADS@SJTU )

44

## StatefulSet

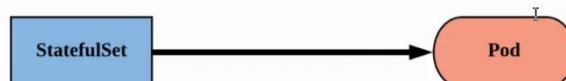
这个和 ReplicaSet 也比较像，但是 StatefulSet 是针对有持久化的对象设计的。这个会更复杂，刚才我们提到 Service 也是一个持久化概念，而 Pod 不是。

- StatefulSet是针对需要有持久化数据(或维护相关状态)的Pod而设计的Workload对象
- 维护的Pod的hostname、网络、存储都会被持久化(persisted)
- 多Replica的情况：每个Pod的名字会有固定的格式和序号：**'<statefulset name>-<ordinal index>'.**



比如Pod的hostname、网络、存储都会被持久化，规定的格式就会比较严格。重启的时候，我们要把持久化数据分配给不同的replica。

- 类似的命名格式同样被用于Pod所对应的网络表示和Volume的标识
- StatefulSet中，Pod的声明周期会按照特定的模式来管理
  - 比如两个Pod，ipads-0和ipads-1，那么在所有情况下，一定是ipads-0先创建起来，再启动ipads-1
  - 类似的，一定是ipads-1先销毁，再销毁ipads-0
- 版本升级通过 **controller-revision-hash** 标签来管理和维护的



\*生命周期

## StatefulSet (3)

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sts-example
spec:
  replicas: 2
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: stateful
    service: app
  updateStrategy:
    type: RollingUpdate
  rollingUpdate:
    partition: 0
  template:
    metadata:
      labels:
        app: stateful
  
```

**<continued>**

```

<continued>
spec:
  containers:
    - name: nginx
      image: nginx:stable-alpine
    ports:
      - containerPort: 80
    volumeMounts:
      - name: www
        mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
    - metadata:
        name: www
      spec:
        accessModes: ["ReadWriteOnce"]
        storageClassName: standard
        resources:
          requests:
            storage: 1Gi
  
```

**<continued>**

首先 kind 不一样，和 replicaset 比的话，我们有一些区别。比如 volumeMounts，因为我们是 stateful 的，我们要 mount 一些文件系统上来。volumeClaimTemplates 是来描述数据卷的，有一些 spec 的要求。

## StatefulSet (4)

- **revisionHistoryLimit**: 和此前的Object类似
- **serviceName**:
  - 对应一个Headless Service

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sts-example
spec:
  replicas: 2
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: stateful
  serviceName: app
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
  template:
    <pod template>
```

此处的 **serviceName** 对应的是 headless service。也就是我们这个 Service 是没有 clusterIP 的。这里为什么要搞一个没有 IP 的 Service 呢？以前我们的 Service，可以用一个 IP 访问多个 Replica。在 Stateful 里，因为会修改存储，K8s 这一层不保证每一层的 Pod 状态是一样的，所以 K8s 选择让每个 Pod 都能被暴露出来让用户去使用、去自己决定管理怎么管理 Pod 的状态，比如用户可以加一层 PAXOS 和 RAFT。

## Headless Service

- 简单来说，没有ClusterIP的Service
- 为什么？
  - Headless Service会为每个具体的Pod分配DNS域名
  - 允许调用者自己去选择和哪个Pod通信
  - Pod直接也可以通过域名和不同序号的Pod通信
  - 适合Stateful的场景 (不同的Pod的状态通常不同)

## Headless Service (3)

<StatefulSet Name>-<ordinal>.<service name>.<namespace>.svc.cluster.local

<pre>apiVersion: v1 kind: Service metadata:   name: app spec:   clusterIP: None   selector:     app: stateful   ports:     - protocol: TCP       port: 80       targetPort: 80</pre>	<pre>/ # dig app.default.svc.cluster.local +noall +answer ; &lt;&gt;&gt; DIG 9.11.2-P1 &lt;&gt;&gt; app.default.svc.cluster.local +noall +answer ;; global options: +cmd app.default.svc.cluster.local. 2 IN A  10.255.0.5 app.default.svc.cluster.local. 2 IN A  10.255.0.2</pre>
<pre>\$ kubectl get pods NAME      READY   STATUS    RESTARTS   AGE sts-example-0  1/1    Running   0          11m sts-example-1  1/1    Running   0          11m</pre>	<pre>/ # dig sts-example-0.app.default.svc.cluster.local +noall +answer ; &lt;&gt;&gt; DIG 9.11.2-P1 &lt;&gt;&gt; sts-example-0.app.default.svc.cluster.local +noall +answer ;; global options: +cmd sts-example-0.app.default.svc.cluster.local. 20 IN A 10.255.0.2</pre>
	<pre>/ # dig sts-example-1.app.default.svc.cluster.local +noall +answer ; &lt;&gt;&gt; DIG 9.11.2-P1 &lt;&gt;&gt; sts-example-1.app.default.svc.cluster.local +noall +answer ;; global options: +cmd sts-example-1.app.default.svc.cluster.local. 30 IN A 10.255.0.5</pre>

这里就展示了不同 Pod 的 IP 的不同，对应了两个不同的 DNS 域名。

## StatefulSet (5)

- **updateStrategy:** 描述了升级Pod所使用的策略，主要有两个合法选项 **OnDelete** 或 **RollingUpdate**。
  - **OnDelete:** 新版本的Pod实例只能在旧版本Pod实例销毁后才能部署
  - **RollingUpdate:** 滚动升级。所有序号比 **partition** 的值大的Pod会按照逆序进行更新
    - 比如，5个replica，mysql-4会先升级，然后到mysql-3, ..., mysql-0

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sts-example
spec:
  replicas: 2
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: stateful
  serviceName: app
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
  template:
    <pod template>
```

updateStrategy 区别不是很大，还是分为全删和滚动升级。但是它会按照逆序进行更新。

## VolumeClaimTemplate

<Volume Name>-<StatefulSet Name>-<ordinal>

```
volumeClaimTemplates:
- metadata:
  name: www
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: standard
    resources:
      requests:
        storage: 1Gi
```

即使对应的StatefulSet被删除掉了，持久化的Volume也不会被自动回收掉。  
必须通过手动删除来销毁这些Volume。

```
$ kubectl get pvc
NAME           STATUS VOLUME          CAPACITY ACCESS MODES STORAGECLASS AGE
www-sts-example-0 Bound pvc-d2f11e3b-18d0-11e8-ba4f-080027a3682b 1Gi   RWO   standard   4h
www-sts-example-1 Bound pvc-d3c923c0-18d0-11e8-ba4f-080027a3682b 1Gi   RWO   standard   4h
```

## 下节课：Kubernetes ( 四 )

- **K8s网络模型**
- **K8s存储(Volume)**
- **从K8s到minik8s**

2022/4/19

今天是我们 K8s 的第四讲，

## 回顾 : Service

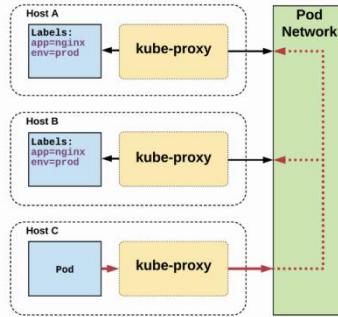
- 主要有四种常见的 Service

- ClusterIP

- NodePort

- LoadBalancer

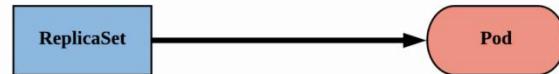
- ExternalName



上节课核心介绍的是 Service，和 Pod 区别是，K8s 的 Service 是长时间提供服务的对象。如果挂掉了，K8s 会把它重启，以保证 Service 能长时间提供对象。最基本的就是 ClusterIP，也就是要给 Service 分配一个固定的 IP，以便于用户能找到这个 Service 使用它。NodePort 就是使用端口来描述。再往下就是建立一个独立的 LoadBalancer 来平衡流量，最后就是 ExternalName 让我们能够访问一些外部的服务。

## 回顾 : Workload之ReplicaSet

- 前面介绍过的ReplicationController(RC)的升级版
- 非常常用的管理Pod备份实例和生命周期的对象
  - 管理Pod的调度、扩容(scaling)、删除等
- 核心任务: 始终保证期望数量(Desired)的Pod实例在运行



这是 RC 的升级版，作为 Pod 的 Replica 的管理者，可以保证期望数量的实例在运行。

## 回顾 : Workload之Deployment

- 在ReplicaSet的基础上，进一步封装的workload对象
- 通过声明式(Declarative)的方式来管理Pod
- 提供了回滚(rollback)机制和版本升级控制
- 版本升级控制通过: *pod-template-hash* 的标签控制
- 每一次升级迭代，会给对应的ReplicaSet和Pod打上新的标签



再往上就是 Deployment，提供了滚动更新的机制。我们还讲了 DaemonSet 和 StatefulSet

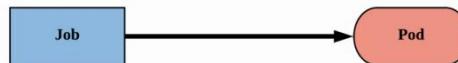
## Outline : Kubernetes ( 四 )

- K8s workload对象: Job&CronJob
- K8s的网络
- K8s存储(Volume)
- 怎么实现一个Service ?

这节课我们继续讲 Job 和 CronJob，接下来我们再将网络和存储。

### Job

- 允许一些简单的run-to-completion的任务
- Job控制器会保证一个或多个Pod执行并顺利结束
- 会一直尝试执行任务，直到其满足了完成的条件 (比如执行100次)
- 即使Job已经完成，对应的Pod也不会马上删除 (需要等待Job本身被删除)



run-to-completion 可能倾向于执行完任务 A 以后立即执行任务 B。它会保证执行顺利结束。我们执行次数或者并发度可能不止一个，我们 Job 可以把多个任务合成一起组成一个 Job。

Service 可能是说，我们有一个长时间的不断运行的服务。当我们请求来了就会立马返回 response。而 Job 就是有 bound 的任务，完成条件达成了以后就自动退出了，它有一个固定的完成上限。Job 本身不会去销毁 Pod，Pod 可能需要手动地去删除。

- **backoffLimit:** Job可以接受的failures的次数 (比如fail 4次就认为这个Job失败了)
- **completions:** 总共需要的成功完成的次数
- **parallelism:** 能够支持的并发运行的Pod的实例数
- **spec.template.spec.restartPolicy:** Job支持的重启策略，目前支持 Never 和 OnFailure 两种

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-example
spec:
  backoffLimit: 4
  completions: 4
  parallelism: 2
  template:
    spec:
      restartPolicy: Never
    <pod-template>
```

它是定义了终止条件的对象，所以有一些条件可以定义在 spec 中，如 backoffLimit。Job 的生命周期可能和 Pod 比较相似，终止条件是完成的次数或者失败的次数。在我们做一些任务系统的时候，接触到的 Job 差不多。最后 Job 是支持重启策略的。

```

Job
$ kubectl get pods --show-all
NAME      READY   STATUS    RESTARTS   AGE
job-example-dvxd2  0/1   Completed  0          51m
job-example-hknns  0/1   Completed  0          52m
job-example-tpkhn  0/1   Completed  0          51m
job-example-v5fvq  0/1   Completed  0          52m
$ kubectl describe job job-example
Name:           job-example
Namespace:      default
Selector:       controller-uid=19d122f4-1576-11e8-a4e2-080027a3682b
Labels:         controller-uid=19d122f4-1576-11e8-a4e2-080027a3682b
Annotations:    <none>
Parallelism:   2
Completions:   4
Start Time:    Mon, 19 Feb 2018 08:09:21 -0500
Pods Statuses: 0 Running / 4 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=19d122f4-1576-11e8-a4e2-080027a3682b
            job-name=job-example
  Containers:
    hello:
      Image:  alpine:latest
      Port:   <none>
      Command:
        /bin/sh
        -c
      Args:
        echo hello from $HOSTNAME!
      Environment:  <none>
      Mounts:       <none>
      Volumes:      <none>
  Events:
    Type  Reason     Age   From           Message
    ----  ----     --   --            --
    Normal  SuccessfulCreate  52m  job-controller  Created pod: job-example-v5fvq
    Normal  SuccessfulCreate  52m  job-controller  Created pod: job-example-hknns
    Normal  SuccessfulCreate  51m  job-controller  Created pod: job-example-tpkhn
    Normal  SuccessfulCreate  51m  job-controller  Created pod: job-example-dvxd2

```

Job 是一种比较简单的对象，比如要求的 completions 是 4，我们启动的时候启动了 4 个 Pod 对象，但是因为 parallelism 是 2，可能执行的时候是两个两个去执行。

## CronJob

- 基于Job的扩展，能够实现一些调度策略
- 用cron的方式来管理调度



Cron 是 Unix 系统里面的经典工具，用于执行周期性的任务。比如我们配置可以每秒 lookup 一下，检查一下 crontabs，看看需不需要当前秒去运行。

ON(8) System Manager's Manual CRON(8)

**NAME**  
**cron** – daemon to execute scheduled commands (Vixie Cron)

**NOPSIS**  
**cron** [-s] [-o] [-x debugflag[,...]]

**DESCRIPTION**  
The **cron** utility is launched by **launchd(8)** when it sees the existence of **/etc/crontab** or files in **/usr/lib/cron/tabs**. There should be no need to start it manually. See **/System/Library/LaunchDaemons/com.vix.cron.plist** for details.

The **cron** utility searches **/usr/lib/cron/tabs** for crontab files which are named after accounts in **/etc/passwd**; crontabs found are loaded into memory. The **cron** utility also searches for **/etc/crontab** which is in a different format (see **crontab(5)**).

The **cron** utility then wakes up every minute, examining all stored crontabs, checking each command to see if it should be run in the current minute. When executing commands, any output is mailed to the owner of the crontab (or to the user named in the **MAILTO** environment variable in the crontab, if such exists).

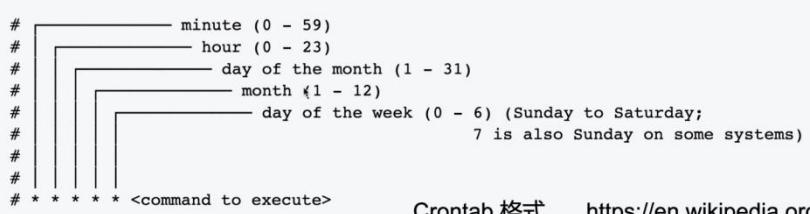
Additionally, **cron** checks each minute to see if its spool directory's modification time (or the modification time on **/etc/crontab**) has changed, and if it has, **cron** will then examine the modification time on all crontabs and reload those which have changed. Thus **cron** need not be restarted whenever a crontab file is modified. Note that the **crontab(1)** command updates the modification time of the spool directory whenever it changes a crontab.

这里也是利用了 Cron 来做调度。

- **schedule:** 任务的调度周期，格式和 Unix 中 cron 工具的 crontab 中的一行是类似的
- **successfulJobHistoryLimit:** 记录的成功完成的 Job 的历史次数
- **failedJobHistoryLimit:** 记录的失败的历史次数

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob-example
spec:
  schedule: "*/1 * * * *"
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  jobTemplate:
    spec:
      completions: 4
      parallelism: 2
      template:
        <pod template>
```

它的核心就是任务调度周期，**schedule** 这里是使用了类似正则表达式的方式来表达的。



每个\*表达的是每个时间上的可读。

Cron Job	Command
Run Cron Job Every Hour	0 * * * */root/backup.sh
Run Cron Job Every Day at Midnight	0 0 * * */root/backup.sh
Run Cron Job at 2 am Every Day	0 2 * * */root/backup.sh
Run Cron Job Every 1 <sup>st</sup> of the Month	0 0 1 * * /root/backup.sh

但是 **schedule** 里还支持一些复杂的 operator。我们就是用这种格式去描述我们的周期，

来决定多久应该执行一次。比起 Job 增加了调度周期的语义。

## K8s Workload小节

- Workload是管理Pod的高层抽象
- ReplicaSet: 支持多备份实例，管理Pod的生命周期
- Deployment: 在ReplicaSet基础上支持版本升级控制等
- DaemonSet: 能够在符合要求的Node上启动类似后台服务
- StatefulSet: 针对状态进行定制化的设计
- Job: 支持短生命周期但是多次执行的任务

Workload 是管理 Pod 的高层抽象。Pod 没有提供备份等功能。Stateful 的情况相当于在 ReplicaSet 上做了一个定制化。Job 就是可以支持多次执行的任务。如果 Job 要求跑 100 次，但是 parallelism 只有 1，如果是 long-running，那这个 Job 就没有意义。Job 更多的是支持生命周期短的。

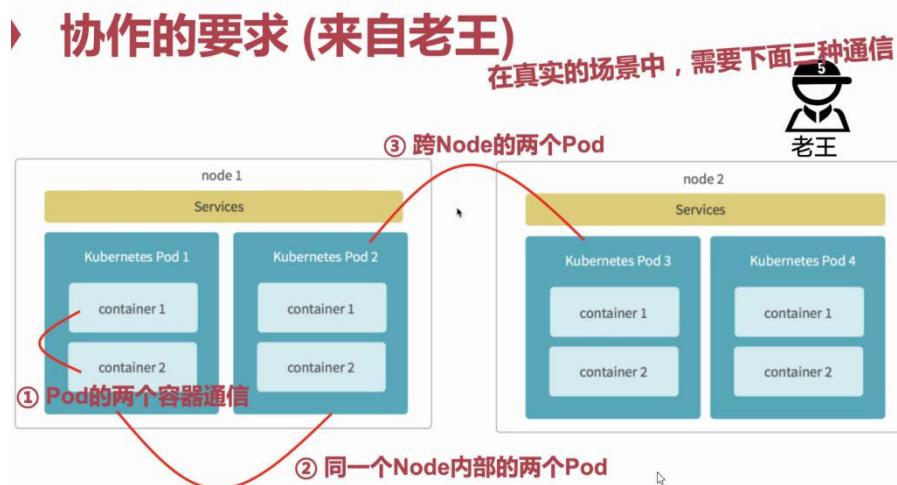
我们这次只要求做了 ReplicaSet。

## K8s 的网络

Refs:

[1] [https://medium.com/@tao\\_66792/how-does-the-kubernetes-networking-work-part-1-5e2da2696701](https://medium.com/@tao_66792/how-does-the-kubernetes-networking-work-part-1-5e2da2696701)  
[2] K8s in Action

我们之前也提过很多次分布式环境了



最后我们部署 minik8s 在一台机器上，可以管理两台别的机器。controller 和 node 也会跑在一台机器上。这样我们就会有 3 个节点来做我们 minik8s 的执行。

协作要求如下：

1. Pod 内部的两个容器通信
2. 同一个 Node 里的两个 Pod 通信
3. 跨 Node 的两个 Pod 通信

## Pod 内容器间的通信

我们必须去做一些网络方面的配置。比如 container1 怎么找到 node2 的服务呢。

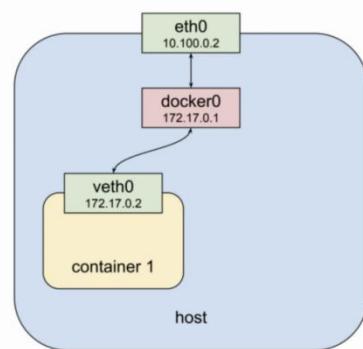
## Pod 内容器间的通信

- Pod 内部的容器应该是共享网络的
- K8s 的目标：
  - 同一个 Pod 内部：所有的容器之间，都应该可以任意地进行通信
  - 基于 localhost
  - 比如，一个 Pod 中有两个容器，一个容器是 Nginx，监听了 80 端口，另一个容器可以通过 localhost:80 去访问到 Nginx

Pod 内部的通信只需要我们去共享网络。所有容器运行在一台机器上，可以基于 localhost 互相进行通信。

## 容器的网络回顾

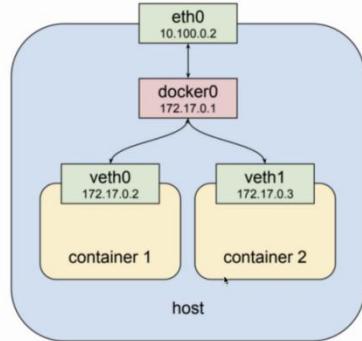
- Eth0：节点的网络接口
- Docker0：网桥
- Veth0：虚拟网络接口
- 容器中的程序通过 veth0 → docker0 → eth0，访问到外部网络



之前我们可能讲过一点，host 里有一个容器的时候，建立一个网桥 docker0，它可以把 container1 的网络接口和 host 的网络连在一起。

## 容器的网络回顾 (2)

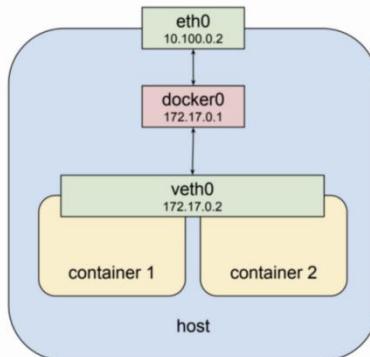
- 两个容器的情况
- Container2的veth1同样连接在docker0上
- 两个容器之间可以通信



容器多了也一样，我们无非新的连接也连到 docker0 这个网桥上。通过这种方法就可以去做通信了。但是这样并没有达到我们之前的目标。虽然容器和容器之间可以通信，但是我们 localhost:80 还是访问了自己容器内的 80 端口，并不是 host 的 80 端口。

## 共享Network命名空间

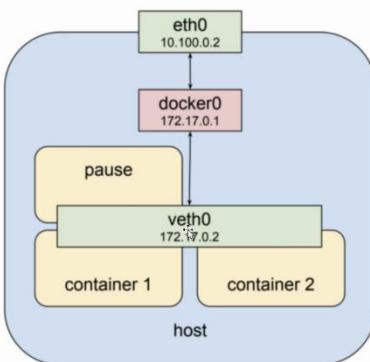
- 通过让两个Container共享一个Network命名空间，可以构成 →
  - 同一个IP地址
  - 两个container不能使用一样的端口



总之容器和容器之间可以共享命名空间，可以认为有一张虚拟网卡，共同连了一个虚拟的线到网卡上。

## Pod内容器间的通信：K8s的实现

- 引入了一个专用的容器，**pause**
- 为Pod中的其他容器提供网络



K8s 还引入了一个专用的 pause 容器，可以把网络管理剥离给 pause 容器。pause 就专门处理网络。在这样之后，网络就搭起来了，达成了我们之前的目标。大家自己在实现的时候，可以不用考虑 pause 的问题。

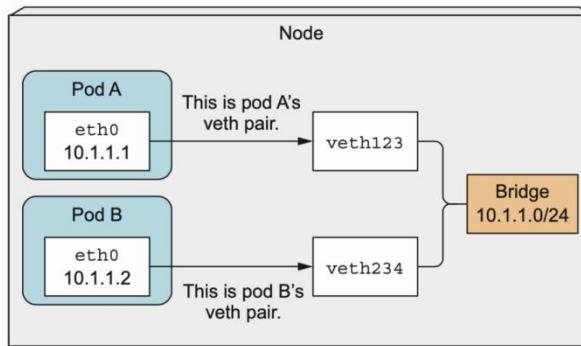
## Pod 间的通信

- Pod 之间可以直接通信 (通过 Pod 的 IP)
- K8s 的目标：
  - 所有的 Pod 之间可以任意通信，且不依赖于 NAT
  - 可以跨 Node 通信，且不依赖于 NAT
  - 一个 Pod 看到的自己的 IP 应该和其他 Pod/Node 看到的 IP 是一样的

Pod 之间怎么通信呢？每个 Pod 自己是有自己的 IP 的，因为我们要通过 IP 去访问这个 Pod。本来 NAT 是可以做一层转化的（端口转化为 IP），这里我们可以不依赖于它。Pod 的 IP 是全局唯一的，别人看到你的 IP 也是统一的。

## Pod 间的通信：相同的 Node 间的 Pod

- 现有的设计可以满足要求

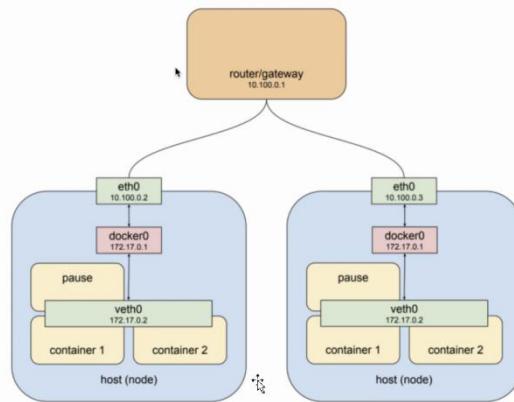


现有的设计看似可以满足要求，一个 Pod 分配一个虚拟网卡，每个 Pod 一个配置一个 IP，连到 Bridge 上去。

## Pod 间的通信：不同的 Node 间的 Pod

- 左边和右边各有一个 Pod
- 它们的 veth0 的 IP 地址都是 172.17.0.2
- 其他 Pod 怎么定位到左边的 Pod 呢？

我们需要扩展下现有设计  
老王

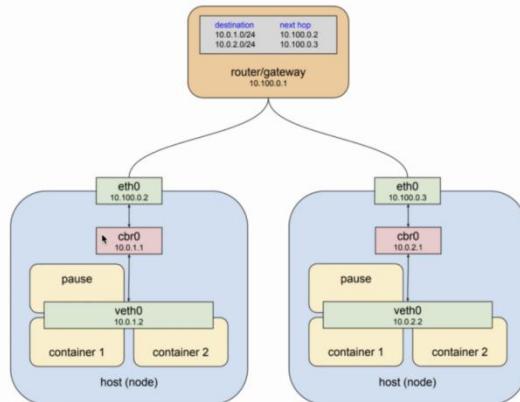


问题是说，我们必须考虑如果我们有多个 Node，我们怎么建立全局一致的 view 呢？以前，我们 node 内部的网络管理没有考虑别的 node 的问题。现在 eth0 可能是连到网关和路由器上的，这两个 eth0 肯定是不同的，在里面大家去初始化自己的网络，可能 docker0 的

IP 和 veth0 的 IP 都是一样的。这样我们右边的 Pod 可能就和左边的 Pod 重名了。所以这里就得进行一个多节点之间的协调。

## Pod间的通信：不同的Node间的Pod

- 全局的IP分配
  - 比如这里会给左边和右边的 bridge 分别分配一个 10.0.1/24 的所有区间，另一边是 10.0.2/24 的所有区间
  - Node 内部的 Pod 会使用这个区间里的一个 IP，比如 10.0.1.2
- 全局的 router/gateway 需要记录对应的路由信息
  - 10.0.1.0/24 的，转发到 10.100.0.2(左边) Node



我们打通外面网关/路由器和里面的 cbr0 网桥。这样我们 IP 的分配是全局的分配，路由器可能预先给分配好。我们加了一些路由规则，就可以为全局的所有的 Pod 定义全局唯一的 IP。通过这样的方法，比如右边的 10.0.2.2 要通信 10.0.1.2。这样我们可以通过 eth0 找到路由器，找到下一跳，再找到 eth0，最后找到 veth0。有了这个东西以后，跨节点的通讯就可以解决了。

## K8s网络



前面介绍的网桥都是谁配置的呢？  
K8s吗？

- Pod 网络
  - 表示整个集群内，Pod 和 Pod 之间通信的网络
  - 由 K8s 的 CNI (Container Network Interface) 插件管理
- Service 网络
  - K8s 集群内存在一段预留的虚拟 IP (Virtual IPs)
  - 这段虚拟 IP 将被 kube-proxy 用来实现服务发现等功能 (service discovery)

Service 的 IP 可以被 kube-proxy 发现。我们本身就让不同的 Service 有不同的 IP，并且存在数据库里，所以它重启以后也是这个 IP。

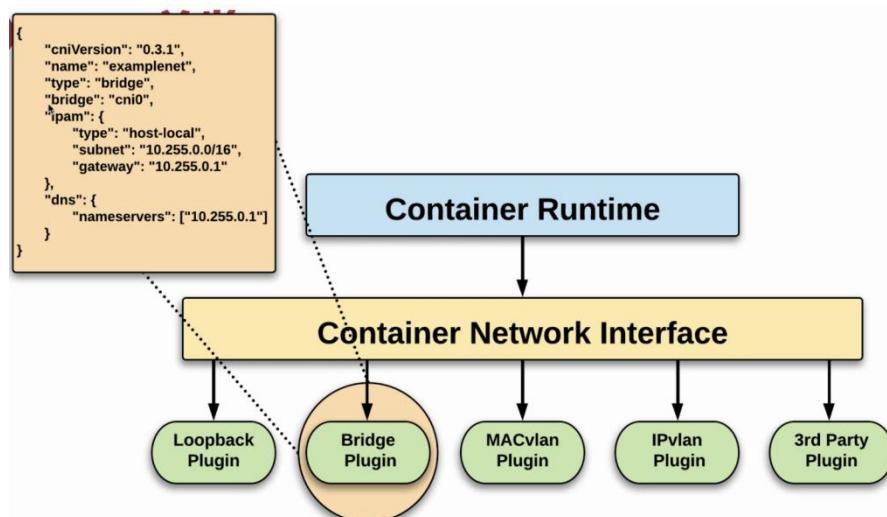
K8s 可能自身没提供网桥的配置功能，它是使用 CNI 插件来配置的网桥。

# CNI 模型（Container Network Interface）

## Container Network Interface (CNI)模型

- Pod网络是通过Container Network Interface (CNI)的抽象实现的
- CNI提供了一系列的函数，来对接容器运行时与一个具体的网络插件实现

- CNI本身目前是CNCF项目
- 基于简单的JSON Schema



## 一些CNI插件的例子

- Amazon ECS
- Calico
- Cilium
- Contiv
- Contrail
- Flannel



- GCE
- kube-router
- Multus
- OpenVSwitch
- Romana
- Weave



网络还是对我们的系统非常重要的。微服务之间都是通过网络相连的。

## K8s网络总结：基础规则

- 同一个**Pod内部**：所有的容器之间，都应该可以任意地进行通信
- 所有的**Pod之间**：
  - 都应该可以任意通信，且不依赖于NAT
  - 所有的**Node**都应该可以和所有的**Pod**进行通信(反之依然)，且不依赖于NAT
  - 一个Pod看到的自己的IP应该和其他Pod/Node看到的IP是一样的

同一个 Pod 内，是共享的网络 namespace，好像是一台机器接了一个虚拟网卡，每个 container 可以认为是一个进程，所以只要网卡是同一张，机器就可以之间互相通信。

Pod 之间，不依赖 NAT，就需要每个人有唯一的 IP，保证可以唯一连到它。

## K8s网络总结(2): 实现与应用

- Container-to-Container**
  - 同一个Pod内部的容器处于一个相同的网络命名空间 (**network namespace**)，并且共享一个相同的IP
  - 可以通过本地网络(*localhost*) 之间在两个容器间通信
- Pod-to-Pod**
  - K8s会为每个Pod分配一个**cluster unique IP**
  - 这个IP在Pod的整个生命周期中都是有效的
  - 要注意的是，Pod本身是易失的

因为 Pod 是易失的，所以 IP 是在生命周期内有效的，Pod 重启之后 IP 可能就会变化。

## K8s网络总结(3): 实现与应用

- Pod-to-Service**
  - K8s会为Service分配一个“持久化的”集群内的IP
  - 通过**kube-proxy** 实现Pod和Service之间的通信/调用
  - Service的IP是持久化的，就是Service对应的Pod挂了也不会变
- External-to-Service**
  - 同样由**kube-proxy** 管理
  - 依赖于云服务商或者是外部的一些实例(比如外部的LB) 来连接外部和内部的服务

我们通过 **kube-proxy** 的方法去通信，它把 IP 换成 endpoint 的 IP 转发过去。ExternalService

也是 通过 kube-proxy，依赖于云服务商进行通信。K8s 能做的就是在 kube-proxy 这一步知道我们下一跳去哪。

## K8s 中的存储

### 计算，网络，然后是？

从实际应用的角度来看，Pod提供了计算抽象，K8s的网络提供了不同场景下的网络连接，除此之外我们还需要什么呢？



是数据的存储！  
小明

接下来介绍存储在 K8s 里怎么做。

## 存储

- 相当一部分的应用需要
  - 在同一个Pod内部的不同容器间共享数据 (不走网络通信)
  - 持久化一些数据，如日志、计算结果等
- 和容器中的卷(Volumes)类似，K8s也通过Volume来提供存储的能力
  - 主要四类：Volumes, PersistentVolumes, PersistentVolumeClaims, StorageClasses.

Pod 需求是多种多样的，有些简单的不需要外部的存储。大多数 Pod 还是需要外部存储的，可能需要记录日志、持久化等功能。K8s 也是通过类似的 Volume 来提供存储的能力，主要有 4 类。

## Volumes

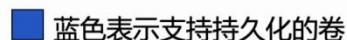
### K8s存储的基础抽象：卷 (Volumes)

- Volume被定义在Pod的Spec中，并且和Pod的生命周期绑定
- 对于单个Pod而言，可以有一个或多个不同类型的Volumes
- Pod内部的任意的容器都可以使用这些Volumes
- 对于Pod意外推出导致的重启，Volume是不会丢失的
- 如果Pod被删除，Volume是否会被保留还是删除，会根据不同的Volume类型而不同

基础抽象还是 Volumes，它定义在 Pod 的 Spec 中，和 Pod 的生命周期绑定。Pod 退出以后，Volume 不一定会被删除。删除与否会和 Volume 类型相关。这也是体现了 Volume 持久化的特性。

### 卷类型: 一些常见的例子

- |                        |                             |                  |
|------------------------|-----------------------------|------------------|
| ● awsElasticBlockStore | ● flocker                   | ● projected      |
| ● azureDisk            | ● gcePersistentDisk         | ● portworxVolume |
| ● azureFile            | ● gitRepo                   | ● quobyte        |
| ● cephfs               | ● glusterfs                 | ● rbd            |
| ● configMap            | ● hostPath                  | ● scaleIO        |
| ● csi                  | ● iscsi                     | ● secret         |
| ● downwardAPI          | ● local                     | ● storageos      |
| ● emptyDir             | ● nfs                       | ● vsphereVolume  |
| ● fc (fibre channel)   | ● persistentVolume<br>Claim |                  |



Volume 对应了比较复杂的后端扩展。容器的 Volume 更多是以 path 规定的。K8s 里的卷对应的是不同的后端的存储。aws 就是对应了亚马逊的存储，gitRepo 可能就是一个网络上可以拉取的镜像。

## 卷 (Volumes)(2)

- **volumes:**

- 一个Volume对象的列表，这些Volume会被挂载在当前的Pod
- 每个Volume对象一定要有一个唯一的name

- **volumeMounts:**

- Pod spec中的容器声明部分
- 表示需要挂载的一系列Volume，通过前面指定的name来找到对应的
- 这里还需要指定挂载的位置：  
`mountPath`

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-example
spec:
  containers:
    - name: nginx
      image: nginx:stable-alpine
      volumeMounts:
        - name: html
          mountPath: /usr/share/nginx/html
          readOnly: true
    - name: content
      image: alpine:latest
      command: ["/bin/sh", "-c"]
      args:
        - while true; do
            date >> /html/index.html;
            sleep 5;
        done
      volumeMounts:
        - name: html
          mountPath: /html
  volumes:
    - name: html
      emptyDir: {}
```

Volume 可能不需要 kind=Volume 去描述，复杂的文件是可以通过 kind=Volume 来单独描述的。它是通过 volumes 字段来描述的，我们虽然说 volume 卷和 container 卷有点相似，还是有点区别的。这样我们就定义了一个空的卷，也可以成为我们的对象。在这里的 volume 更灵活一些，可以定义空卷，在后面挂载上来的容器就可以做一些共享的读写和并行的工作。

刚才讲了 volume 的一些定义。包含 volumes，有唯一的名字挂载进当前的 pod。之前的 Pod Spec 中已经有 volumeMounts 这个东西了。在这里我们就是创建了这个容器里的文件夹，我们可以自由去使用它。

## 持久化卷 (PV)

### 持久化卷 (Persistent Volumes, 简称PV)

- 和Volume类似，PV是另一类存储资源/对象
- 和Volume的区别
  - PV是整个Cluster层面的资源 (不属于任何的namespace)
  - PV是底层的具体的存储资源的K8s对象抽象
  - 底层具体的资源可以包括NFS, GCEPersistentDisk, RBD etc.
- PV通常是由集群的管理员来维护和提供的
- PV的生命周期和Pod是独立的
- PV不能直接挂载在一个Pod上，需要通过另一个对象  
**PersistentVolumeClaim (简称PVC)**

Volume 之上第二个类型就是持久卷。Volume 对象相对简单，也不指定存储介质和存储服务。而 PV 是 cluster 层面的资源，它不属于任何的 namespace。持久化需求可能是跨越 namespace 的，所以要求持久化和 namespace 解耦。它是底层存储资源的抽象，比如 cephfs 这种分布式文件系统的抽象，这里的 PV 就对应了这些资源。

这里的问题是说，既然是全局管理员提供的，那么生命周期肯定不会和 Pod 绑定。所以

PV 不会直接挂载在 Pod 上。PVC 负责抽象一层 PV，Pod 挂载的实际是 PVC。

## PVC

### PersistentVolumeClaims (PVC)

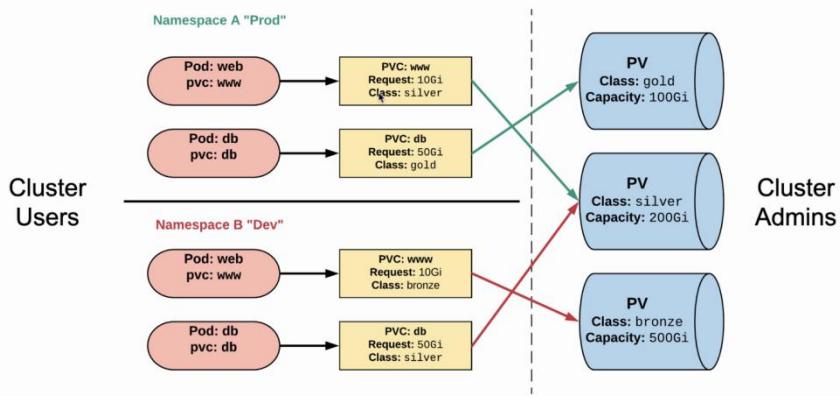
- 和持久化卷(PV)不同，PVC是被包括在某个namespace内部的
  - 被Pod直接使用
- 将Pod的存储需求映射到真实的存储资源上

这也是为什么用Claim这个词的原因，  
PVC表达了Pod对于存储的要求/主张



用户对 Pod 的 Persistent Volume 的需求宣称。K8s 把需求映射到真实的存储资源上。

### 持久化卷与Claims



我们的 PV 是独立于 namespace 存在的。所有 namespace 都可以使用这个 PV。Pod 创建一些 PVC，和 Pod 绑定。K8s 根据 Class 再把 PVC 和 PV 绑定。

我们假设没有 PVC，直接在 Pod 设置 Request 和 Class 呢？PV 是一个全局的概念，管辖的范围很多，如果直接写 PV 的话，管理起来有点麻烦。

## 小明的疑惑



为什么不直接使用PV呢？

小明

- 一个原因是为了解耦，PVC通常只表达Pod的存储需求，但是具体使用什么PV（比如是NFS、Ceph等）是Pod自己不太关心的
- PVC表达需求，PV是真实的实现（这也是K8s声明式的一个体现）
- 这样还能保证Pod的Claim在底下环境发生变化后仍然是能支持的（如NFS换成了其他存储）



老王

后端的存储 Pod 不应该太关心，这是比较关键的。Pod 的定义的 PVC 只有名字，PVC 直接向 PV 进行连接就行了。

## 持久化卷 (2)

- capacity.storage:** 表示存储的总的容量
- volumeMode:** Volume的类型，包含两种：**Filesystem** 或 **Block**。
- accessModes:** 对于该Volume，支持的访问方式。使用中，通常包含下面的可选项：
  - ReadWriteOnce
  - ReadOnlyMany
  - ReadWriteMany

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfsserver
spec:
  capacity:
    storage: 50Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /exports
    server: 172.22.0.42
```

它里面主要包含存储的总容量、类型（文件系统还是 Block（裸硬盘））和 accessModes 的权限管理。Once 和 Many 是指允许同时使用的 Pod 数量。

## 持久化卷 (3)

- persistentVolumeReclaimPolicy:** 表示当关联的PVC被删除后，PV需要执行的操作，可选项包括：
  - Retain – 保留数据，后续手动清理
  - Delete – 马上清理数据
- storageClassName:** 可选项。可以通过这个选项指定当前PV对应的storageclass。PVC只有包含同样的storageClassName时才能关联到该PV
- mountOptions:** mount选项（和Volume类型相关，可以简单理解为Linux `mount`时的选项）

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfsserver
spec:
  capacity:
    storage: 50Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Delete
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /exports
    server: 172.22.0.42
```

`storageClassName` 这样和我们 PV 里的 Class 是对应的。`mountOptions` 会和我们的 Volume 类型相关。这些是持久化卷包含的字段，比前面的 Volume 复杂很多。

PV 和 PVC 里定义的很多字段是一致的。

## PV与PVC

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-selector-example
  labels:
    type: hostpath
spec:
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteMany
  hostPath:
    path: "/mnt/data"
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-selector-example
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      type: hostpath
```

## 持久化卷(PV)的几个状态



比如我们没有用 `delete`，用的是 `retain`，就是可以被回收的状态。这里其实就讲完了 PV 和 PVC。

## StorageClass (存储类)

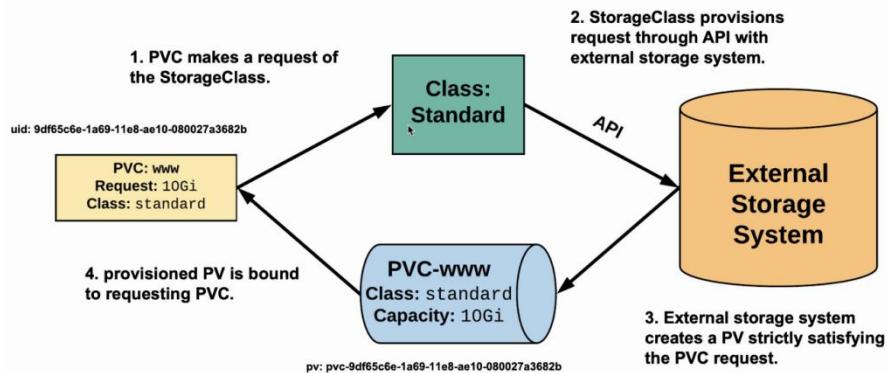
- 前面介绍过了`storageclassname`
- 其实，存储类 (StorageClass) 是一个单独的K8s对象
  - 基于外部持久化存储资源(PV)的一个更高层抽象
- 为什么需要提出一个存储类？
  - 云上已经有了大量的提供存储的服务，存储类能够结合这些存储服务提供：动态的集群内存储
  - 避免了管理员手动创建PV

接下来是 StorageClass，为什么会有这么的一个存储类呢？这是因为 PV 也是有一些问题的，PV 是 cluster-level 的一个抽象，后来大家发现这个抽象也不是这么好，提出了一个新的存储类。

因为云上大家有自己的云存储服务：

我们通过 PVC 创建 request，K8s 通过 API 找 storage system 生成一个 PV，要求满足 PV 的请求，然后再返回。这个 PV 是由 K8s 自动创建的。

## 存储类 (StorageClass) (2)



在这样的一个结构下，存储类才是一个关键的抽象。PV 是 K8s 动态的创建出来的。

**Q:** 以前 PV 才是核心，在这样的一个场景下为什么在这个结构上还要提一层外部存储类。有什么是 external storage system 相对于自己的 PV 好实现的呢？为什么要把 PV 的功能收回掉，再提出一个新的 storage class 呢？

**A:** PV 是有容量限制的，一开始可能针对的是静态的存储，现在的存储是向着动态发展的。现在云存储都是按需付费的。在这种情况下，PV 管理起来就会比较麻烦了。

## 存储类 (StorageClass) (3)

- provisioner:** 定义真正提供外部存储的实体
- parameters:** 指定针对于 provisioner的一些配置参数和选项
- reclaimPolicy:** 当对应的PVC被删除后，底层的存储的处理方式
  - Retain – 手动清理
  - Delete – 由provisioner来清理数据

```

kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: standard
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  zones: us-central1-a, us-central1-b
reclaimPolicy: Delete
  
```

这里有个 zones，就是美国中部 1-a 机房，这就很像云存储的配置了。

到此为止，K8s 基本上就差不多了：计算、网络、存储、核心对象等东西。大家做的内容都包含在其中了。

## 怎么实现一个 Service？

### 回顾：Service

- 每个Service有自己的静态 IP 和端口
- Client (比如一个Pod) 通过访问Service的IP和端口来调用服务
- Service的IP是虚拟的
  - 没有被分配给任何具体的网络接口，无法被ping
- K8s的Service:
  - 一个虚拟IP地址+端口
  - 映射到真正的一些endpoint (Pod)
  - 这个映射是透明的

Service 自己是有静态 IP 和端口的。这些方法都是通过 kube-proxy 转移，最终变成一个可以访问的服务。这个 IP 不是一个实际的互联网服务，我们没办法去 ping 它。它并没有被分配给实际的网络端口。所以说，我们要做一个 K8s Service，我们需要搞到虚拟 IP 地址和端口，映射到一些 endpoint 上。这个映射对于用户是透明的，用户只知道 Service 的 IP。

### Kube-proxy是实现Service的关键

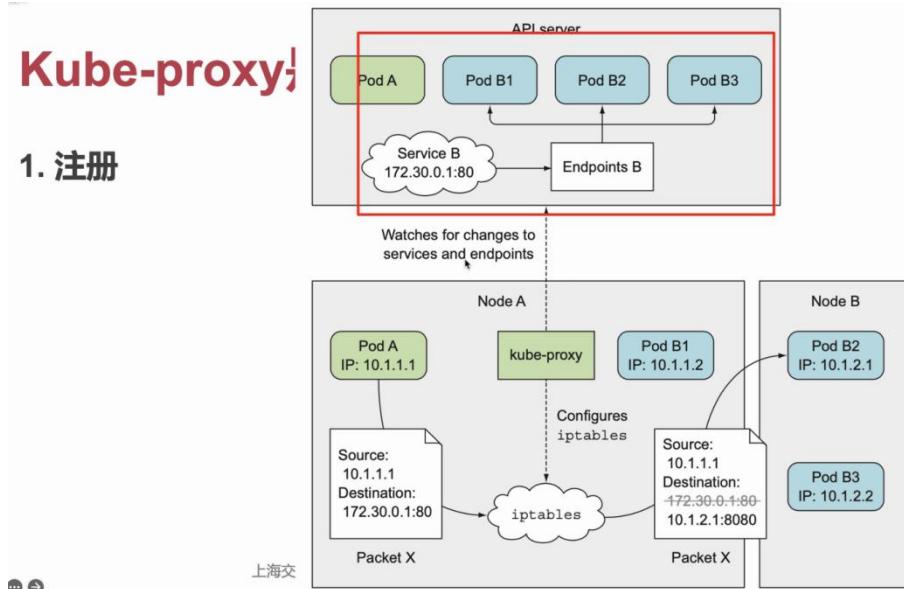
- 当注册一个新的Service的时候，Service的信息会被注册到API Server中
- API Server通知到每个Kube-proxy，让每个Kube-proxy记录下新的service的信息
- Kube-proxy在本地的Node中，通过iptables，来过滤所有的和service的IP:port对应的访问
- Kube-proxy将拦截到的请求转换成真正的EP的地址

我们有 proxy，才能把虚拟 IP+端口转化成真正的 port。比如我们拿到虚拟 IP 10.0.1.2，proxy 会变成 Pod 的真正的 IP。

API Server 是控制面的 Server。API Server 建立了 Service 的 IP 和端口到名字的映射。记录之后，我们就要改 iptable，改了之后，就可以建立了 IP 到 port 之间的监控。

# Kube-proxy

## 1. 注册



我们要注册 Service B，建立了 endpoint B 的连接。Pod A 尝试往 172.30.0.1:80 发请求，这个 IP 是不存在的，只是注册到了 API Server 上了。这个核心就是去找 iptable，iptable 会配置这个请求，如果见到 172.30.0.1:80，就自动转发到 10.1.2.1:8080 上。这个规则就把对应 Service 的虚拟 IP 请求转发成了真实的 Pod 的 IP。它只是配置了这台机器上的 IP 的表。

# Kube-proxy

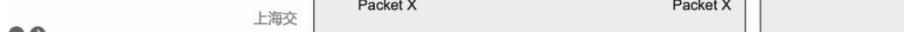
## 1. 注册

## 2. Pod-A发起请求

## 3. Iptable拦截

## 4. 修改dst地址

## 5. 请求到达B2



到达 Node B 之后，访问就结束了。这个就是今天的内容。

## 下节课：Kubernetes ( 5 )

- 从K8s的内部实现到minik8s
- 其他编排系统案例：Borg设计

## Reference for Lab (1)

- **Kubectl**: <https://kubectl.docs.kubernetes.io/guides/>
- **K8s CNI**:
  - <https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/>
- **K8s CSI**:
  - <https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/>

2022/4/22

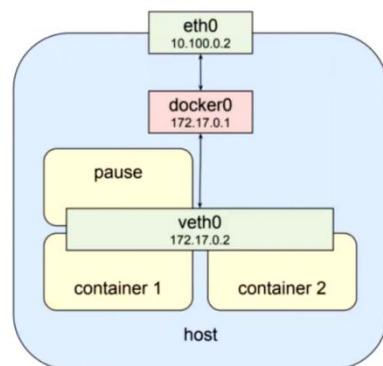
上节课我们把 K8s 核心的功能已经讲的差不多了。首先是讲了 Workload

## 回顾 : K8s Workload

- Workload是管理Pod的高层抽象
- **ReplicaSet**: 支持多备份实例，管理Pod的生命周期
- **Deployment**: 在ReplicaSet基础上支持版本升级控制等
- **DaemonSet**: 能够在符合要求的Node上启动类似后台服务
- **StatefulSet**: 针对状态进行定制化的设计
- **Job**: 支持短生命周期但是多次执行的任务

## 回顾 : K8s Pod容器间的通信

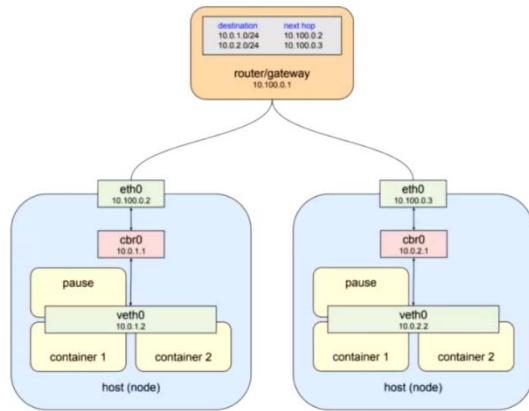
- 引入了一个专用的容器，`pause`
- 为Pod中的其他容器提供网络
- 基于Linux网络命名空间



之后我们就介绍了容器间通信，让 Pod 共享一张虚拟网卡，和网桥链接，最终可以连到 Host 机器的网卡上，最终可以和外界通信。`pause` 容器实现的时候并没有那么重要，`pause` 初始化网卡，这样容器 1 和容器 2 就不需要初始化网络了。这是同一个机器上 Pod 中的容器的通信，

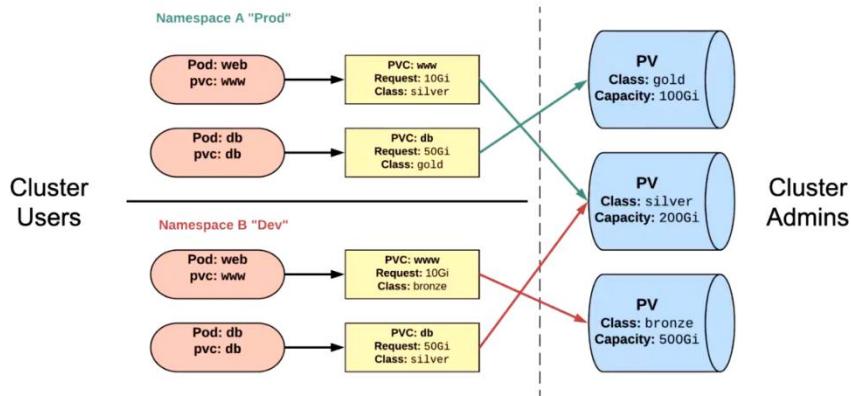
## 回顾：Pod间的通信

- 全局的IP分配
    - 比如这里会给左边和右边的bridge分别分配一个 $10.0.1/24$ 的所有区间，另一边是 $10.0.2/24$ 的所有区间
    - Node内部的Pod会使用这个区间里的一个IP，比如 $10.0.1.2$
  - 全局的router/gateway需要记录对应的路由信息
    - $10.0.1.0/24$ 的，转发到 $10.100.0.2$ (左边) Node



如果要跨节点的话，每个 Pod 要有独立的 IP，所以每个 Pod 要能够被 global 看到。这里的 IP 如果要求全局可见且唯一，我们就必须在路由器和网关上加路由表之类的东西，把下一跳的位置给定下来。如果有更多的节点相连，路由器知道下一跳去哪就行。

## 回顾：持久化卷(PV)与Claims(PVC)



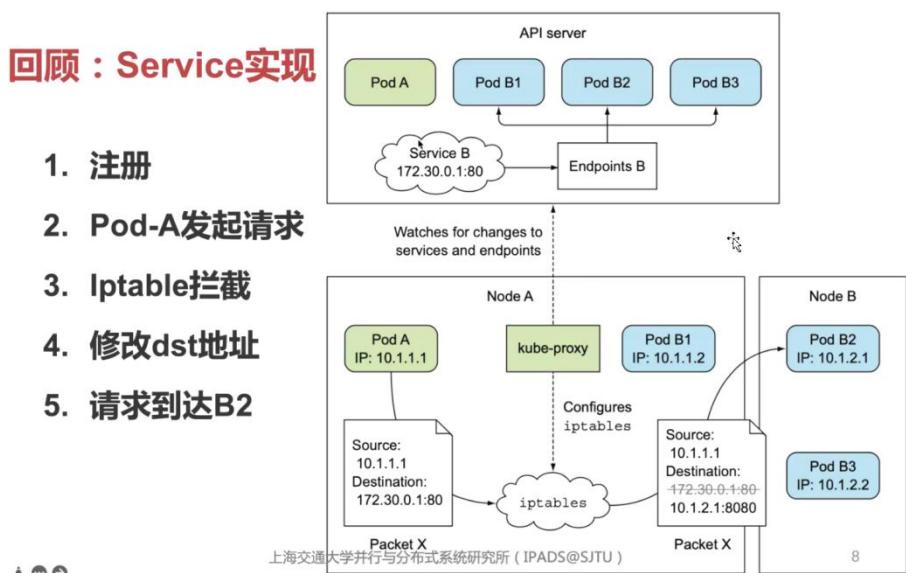
PV 这里的持久化卷，会描述一些卷本身的形态。尤其是比较早期的时候，管理者会拿 PV 对应一个真实的云服务。这样的话，我们通过这种方法，把用户的需求和实际存储的形态给进行解耦。PV 和 PVC 可以通过映射的关系来进行使用。PV 是静态的单元，而存储类是相对动态的，我们可以用 `undemand` 的方式来生成一个 PV，对应我们的 PVC。一种存储可能只有一个存储类，不需要再手动创建 PV 了，并且支持了动态扩充/缩减的特征。

# 回顾：基于Kube-proxy实现Service

- 当注册一个新的Service的时候，Service的信息会被注册到API Server中
- API Server通知到每个Kube-proxy，让每个Kube-proxy记录下新的service的信息
- Kube-proxy在本地的Node中，通过iptables，来过滤所有的和service的IP:port对应的访问
- Kube-proxy将拦截到的请求转换成真正的EP的地址

最后我们讲了怎么实现一个Service。在注册一个新的Service的时候，就会注册到API Server中。kube-proxy就会记录Service的信息，其实就是通过iptables的forward等功能，把原本的Service转化成真正的Pod的IP地址。我们拿到IP以后真正的和服务建立联系。这里的代理还是比较重要的，ICS中的proxy可能就是说client<->proxy<->server。当时我们的proxy没有做太多包的修改。

而Kube-proxy是一个本地的proxy，它不会做简单的转发，而它是修改了全局配置表iptables从而做拦截和转发，并不是直接通过kube-proxy纯软件的拦截和转发，而是通过OS的网络机制做的拦截和转发。



上节课最后我们介绍了Service怎么实现。核心就是当有一个Service注册到API Service的时候，要记住IP和端口，告诉kube-proxy新的服务有什么服务有什么端口。Service的本身IP和端口还有各个endpoint都要告诉kube-proxy，kube-proxy根据这些信息配置iptables，然后就可以做到自动的拦截和转发。

# Outline : Kubernetes ( 5 )

- 从K8s的内部实现到minik8s
  - K8s组件与交互方式
  - K8s中的etcd的作用
  - K8s中的API Server的功能与实现
  - K8s中的Scheduler的设计
  - K8s中的Kubelet的功能与设计

今天的核心内容就是说怎么从 K8s 的内部实现到 minik8s，我们之前还是以 K8s 的功能介绍为主。讲了这么多散的东西，那么怎么拼在一起呢？

## K8s的内部实现



小明

我已经对K8s提供的一些能力，以及应用层面的应用比较熟悉了。  
可是，我还是对怎么实现minik8s有点虚.....



老王

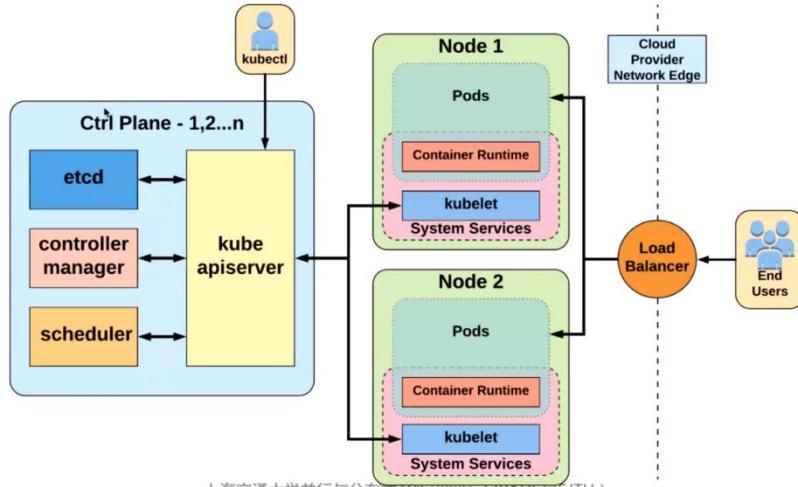
哈哈别怕，我们今天会深入讲一下，k8s  
中的系统内部实现

可能现在是大家比较关注的问题。落实到实现，中间还是有些距离。大家如果对实现有问题，可以问老师和助教。

## K8s 的组件及其交互方式

整体架构我们之前也看到过，可以分为控制面和数据面。

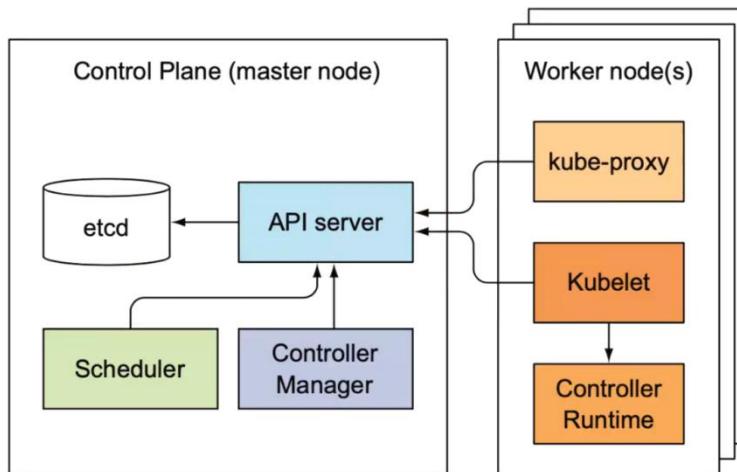
## K8s的架构



但是所有的节点里还是会做一些控制的事情。master 里有四个模块，APIserver、etcd、controller manager 和 scheduler。用户这边还有一个 kubectl，可以创建、获取、修改一些 API 对象。

节点部分是做实际的处理，包含 kubelet 和 API Server 相连的。Pods 里面包含 container runtime。外面还有终端用户，不关心有没有 K8s，只是把请求发进来。

## K8s的架构 : miniK8s的一个参考架构



我们可以参考这个架构去实现。但是不强制一定要按照这个功能实现。在这个实现下，和控制相关的，全部指向 API Server，此时我们的控制面就是一个中心化的，把 API Server 当做枢纽。

# 组件之间怎么交互的？

- 所有的k8s组件，都只和API server交互
  - 当然，API server自己除外
  - 组件之间甚至不感知到其他组件的存在，比如Scheduler不知道ControllerManager的存在
- API Server是系统中唯一和etcd交互的组件
  - 其他组件只通过API Server去访问/修改 etcd中的数据



在这种情况下，组件相互之间是感知不到的，它们只和 API Server 交互。大家都是基于对象在操作，对于其他进程的实现和使用不是特别了解。

etcd 里保存持久化数据，所以它只被 API Server 操作。API Server 忠实地把大家对持久化数据的操作保存在 etcd 中。

## 集群状态和 etcd

etcd 是这些组件里最没那么 “K8s” 的一个结构。

### etcd

- Etcd是一个独立的开源软件，被K8s用来存储集群数据
- 键值对存储 (Key-value store)
- 核心特性：强一致性、高可靠的存储



CSE 的时候讲了 PAXOS 和 RAFT，可以认为它们分别对应了 zookeeper 和 etcd。zookeeper 出现时间可能比 raft 还要早。我们用自己的 kv 做也可以，但是最后要展示可靠性。etcd 在这里就是单纯的应用了，被 K8s 拿过来做持久化数据的管理。

## etcd

- 存储K8s中的对象信息

- 更具体点说，整个K8s集群中的所有的有持久化需求的信息
- 持久化要求：K8s集群崩溃后，重启时能够恢复的状态

- K8s支持etcd2 和 etcd3

- Etcd3的性能更好，但是etcd2的格式相比3更干净
- Minik8s中大家可以使用etcd，也可以选用其他的kv-store (甚至自己写)

我们需要保证集群的 Crash Consistency。之前我们学的更多的是并行的一致性，这里强调的是崩溃一致性。任意一点都可能发生 crash，重启之后的软件能不能继续为用户提供服务。

## etcd: K8s的数据是怎么存储的？

- K8s把所有的数据都存放  
在etcd的/registry目录下
- 左边给了一个案例，我们  
可以通过etcdctl来查看  
registry目录下加的内容

```
$ etcdctl ls /registry
/registry/configmaps
/registry/daemonsets
/registry/deployments
/registry/events
/registry/namespaces
/registry/pods
...
```

展现出来的可能还是基于文件系统的。存什么数据是 K8s 自己决定的。我们看到之前学到的很多 API 对象都在里面，因为 API Server 直接和 etcd 连接，所以里面存了很多的 API 对象。

## etcd: K8s的数据是怎么存储的？

- 以Pod为例，我们可以进  
一步查看Pod目录下的信  
息

```
$ etcdctl ls /registry
/registry/configmaps
```

- 可以看到这里是Pod对  
的命名空间

```
$ etcdctl ls /registry/pods
/registry/pods/default
/registry/pods/kube-system
```

- Default是系统默认的命名  
空间
- Kube-system是控制面的命  
名空间

```
/registry/pods
```

```
...
```

我们进入到 Pods 里面，它就有不同的 namespace，比如 default 和 kube-system (master 相关的 pods)。再往里面就是一些具体的 Pod 的，会有对应的 uuid。

- 更进一步，`default`下面

对应是具体的Pod

```
$ etcdctl ls /registry  
/registry/configmaps
```

- 这里有3个

```
$ etcdctl ls /registry/pods
```

```
$ etcdctl ls /registry/pods/default  
/registry/pods/default/kubia-159041347-xk0vc  
/registry/pods/default/kubia-159041347-wt6ga  
/registry/pods/default/kubia-159041347-hp2o5
```

- 选择其中一个Pod，查看

里面的信息

```
$ etcdctl ls /registry  
/registry/configmaps
```

- 这里是Pod的配置信息

```
$ etcdctl ls /registry/pods
```

```
$ etcdctl ls /registry/pods/default
```

```
$ etcdctl get /registry/pods/default/kubia-159041347-wt6ga  
{"kind": "Pod", "apiVersion": "v1", "metadata": {"name": "kubia-159041347-wt6ga",  
"generateName": "kubia-159041347-", "namespace": "default", "selfLink": ...}}
```

我们get它的信息，就可以得到一个json格式的信息，Pod作为API对象。它里面的信息都被保存在了etcd中。核心就是以目录的方式来组织对象信息的。

## etcd: K8s的数据是怎么存储的？(2)

- Etc按**照目录的方式**组织整个集群中的对象信息
- 最终节点保存的是对象的完整Spec，以及它们的运行时Status**

最终保存的是完整的Spec（静态信息）和运行时的状态。运行时的状态的例子就是一个Service，它是一个固定的IP，重启之后还是原来的Service。所以这个Status就应该放到etcd中对应的Service的IP信息中。

etcd为了保证并行的一致性。API Server可能是多线程的结构，需要同时应对多个模块中的并行请求，所以etcd需要支持并发请求。如果我们对同一个对象进行修改，就会出现data race。保证并行一致性的方法就是OCC。

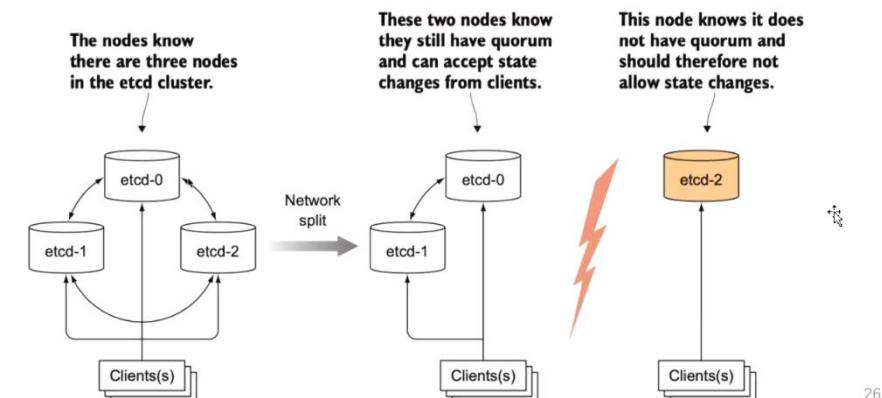
## etcd: 数据的一致性和有效性(CONSISTENCY & VALIDITY)

- OCC (optimistic concurrency control , 或optimistic locking)
  - 当对数据进行并发修改时，OCC不适用lock，而是通过版本号(Version)来管理
  - 每当一个数据修改时，对应的Version会增加
  - 一个Client去修改数据时：
    - 读取数据以及对应的Version
    - 修改数据，提交更新的数据和之前读取的Version
    - **如果API Server或者etcd发现Version和当前数据的最新Version不一致，说明中间发生过别的修改，这一次操作失败**
    - Client需要重新读取数据并进行修改
  - **效果**：当两个并发的Client要修改一个对象时，只有第一个会成功
  - 每一个K8s对象有一个metadata.resourceVersion的项，通过这个来检查对应的对象的版本

核心就是把 Version 进行操作，每次操作完了进行对比。第一个 commit Version 的人会成功，第二个人发现了这件事情就要重试。如果大家实现自己的 KV Store 的话，要注意 API Server 是可以并行操作 KV 的。

## etcd:数据的一致性和有效性，出错了怎么办？

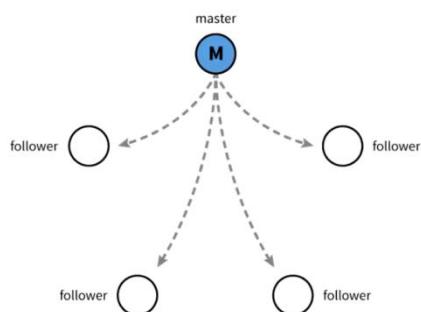
网络出现分裂的时候，只有Majority的一侧能够更新etcd内部的状态，另一层会停留在旧的版本状态下，无法更新



26

## etcd: 怎么保证Majority这点呢？

内部实现：“Raft Consensus”  
among a quorum of systems  
to create a fault-tolerant  
consistent “view” of the  
cluster.

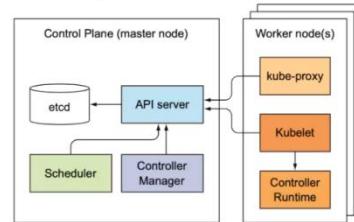


<https://raft.github.io/>

Image Source

## 之前的问题：为什么只有API Server能访问 etcd？

- 如果多个组件都能直接访问etcd的话，每个组件都需要实现对应的Optimistic locking的机制
- K8s中
  - 只需要在API Server中实现Optimistic locking
  - 保证数据的一致性
  - 简化实现复杂度



Q: 为什么我们设计让所有组件都通过 API Server 来访问 etcd 呢？

A: 因为如果大家都能访问 etcd，这件事情就会很复杂。etcd 所有相关持久化的操作，都要做 OCC。其他机器上的模块，如 kube-proxy 和 kubelet 要连到 etcd 就会很麻烦。去中心化的好处就是大家互相之间的开销就小了，API Server 不容易出现瓶颈，但是分布式的管理实现就会复杂。

## API Server

前面讲了 etcd，核心就是大家知道这个是什么东西、知道它怎么用就行了。

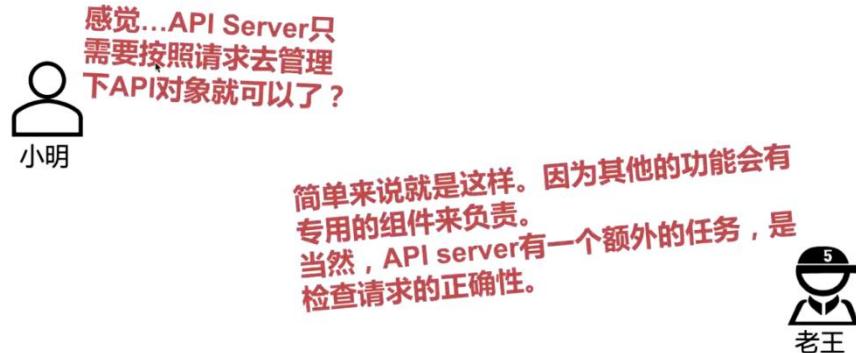
API Server 就是我们集群的核心，我们认为它是一个 CRUD（增删改查）的提供者。

## 什么是API Server？

- API Server是整个K8s集群的中心
- 通过Restful的接口，提供了对集群内部对象的CRUD操作
  - Create, Read, Update, Delete：查询、创建、修改、删除集群内的API对象
- API Server把所有的数据都存放在etcd中
- Optimistic locking: 保证并发更新的一致性

API Server 其实就是提供了增删改查去操作 etcd 中的对象。API Server 并不关心 Pod 和 Service 本身，它只关心怎么创建、怎么修改。API Server 只负责对象的维护。

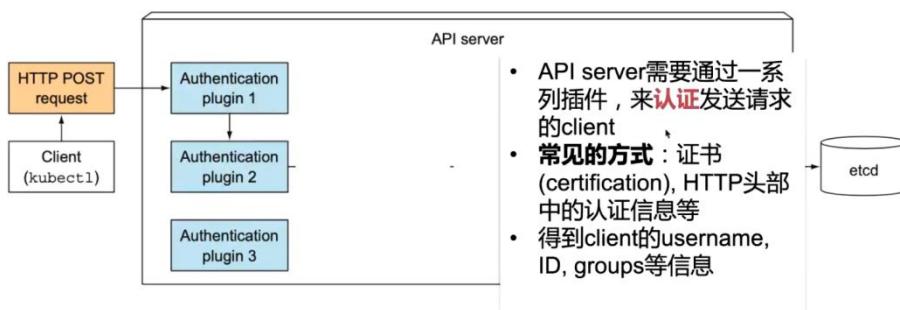
## 什么是API Server？（2）



API 对象对应一个真实的服务/Pod，就需要其他模块来完成。API Server 是唯一和外界（kubectl）进行联系的。kubectl 是管理者来管理应用的客户端，肯定是要发送各种各样的请求过来的，它还需要检测请求的正确性，并且看看每个人有没有权限去做这些事情。主要分成三步。

第一步叫认证，通常是在 request 请求中提供一些认证的信息。（证明自己是谁）

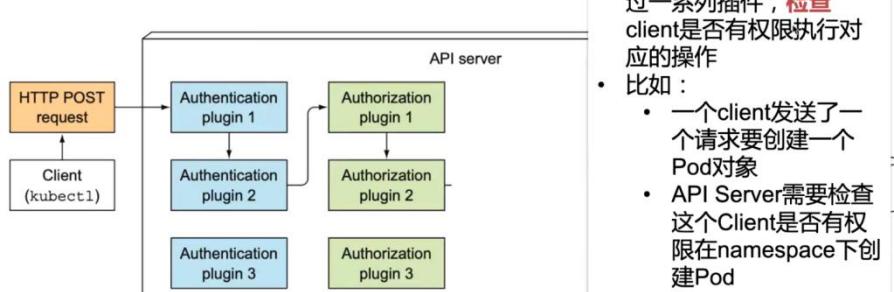
### • API Server的Validation流程



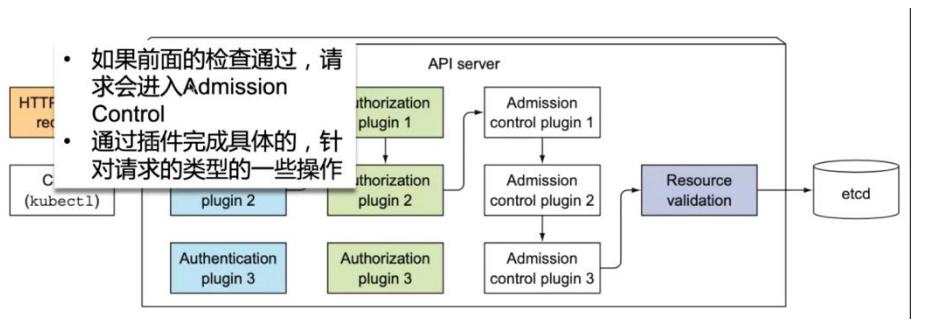
这些信息可能并不包含在 request 请求里，可能存储在我们的 etcd 中，通过这些信息我们完成第一步认证。

第二步是授权，也就是检查 client 有没有权限执行对应操作。

### • API Server的Validation流程



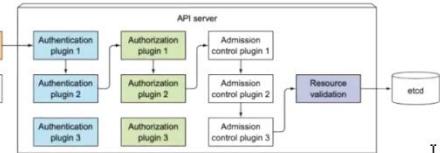
第三步叫做 admission control，也就是我们检查完了一些权限之后，我们要做具体的操作了，在这个过程中我们还是要做一些控制。



Admission Control 会有一些例子，比如 AlwaysPullImages，就是说当我们重新部署一个 Pod 的时候会重新拉取一次镜像。这种情况通常是因为我们的 image 更新很频繁的情况。还有一个例子是 NamespaceLifecycle，如果发现 namespace 不存在/被删除，就会拒绝掉这个请求。之前我们的检查是我们的创建权限，而这里就是针对具体参数的检查，比如 namespace 是否存在。这个检查更加的细粒度。

- **API Server的Validation流程**
  - Authentication → Authorization → Admission Control
- **Admission Control插件的例子**
  - **AlwaysPullImages**: 把Pod的Spec中的imagePullPolicy强制修改为 **Always**，这会使得Pod每次被部署的时候都会重新拉一次镜像 (而不是使用本地缓存的镜像)
  - **NamespaceLifecycle**: 如果Pods所指定的Namespace正处于被删除的状态，或者不存在这个Namespace，这个插件会拒绝掉这个请求
  - 还有很多其他的例子[1]

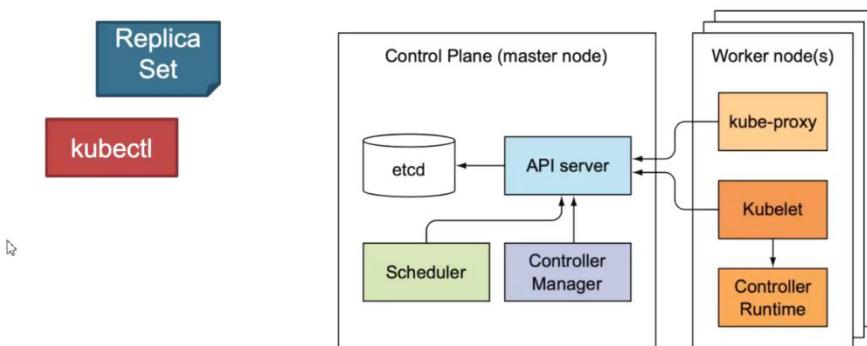
[1]<https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/>



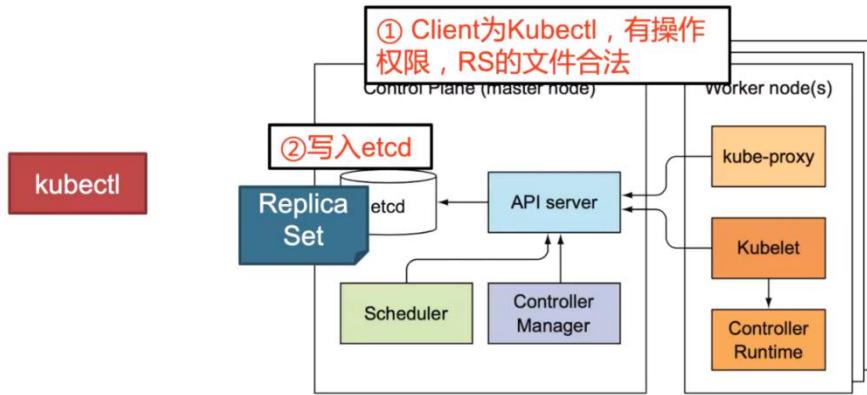
接下来我们来看一个例子：

## 案例

- 案例：当kubectl创建一个ReplicaSet的API对象时



假设我们要创建一个 Replica Set，我们通过 kubectl 创建这个请求。



然后我们检查有没有创建 Replica Set 的权限，第三步就是检查 Replica Set 是否合法，比如文件本身语法有问题，或者 Replica set 设计了一些没有权限访问的数据，比如 Replica Set 指向了一些没有权限访问的 Pod。完成了这三步之后，API Server 就创建了这个 Replica Set，并且把对应的信息存到 etcd 中。

## 小明的困惑

- 案例：当kubectl创建一个ReplicaSet的API对象时



就算API Server本身不做任何的事情，但是其他的模块  
怎么能够知道什么时候需要开始操作呢？

小明

API Server 把 Replica Set 创建之后，总得有人把 Replica Set 实际的语义做处理。比如我们的 Replica Set 的 Spec 中期望 Pod 数量是 3。这个其实就通过 Controller Manager 去负责这件事情。而 API Server 只是完成了对象的创建/更新/删除。



这是一个很好的问题。  
API Server虽然只维护和API对象相关的  
管理，但是提供了一个关键的机制：  
Watch

老王

Controller Manager 怎么知道什么时候要实现 Replica Set 的语义呢？其实就是 Watch(监控) 机制。

# API Server的监听机制

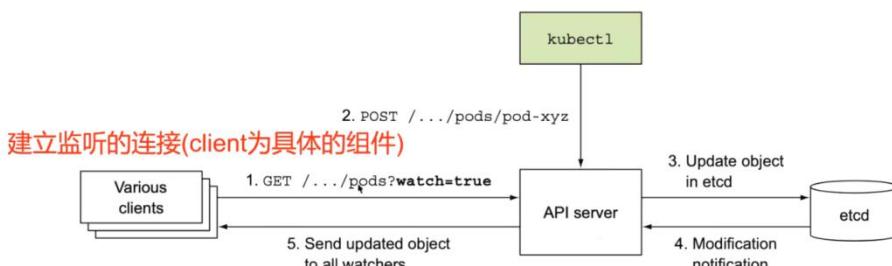
- 在创建ReplicaSet的案例里，真正做RS的操作的是RS的 controller (后面课程会介绍😊)
- API Server监听机制：控制面的组件可以要求API Server在特定(类型)的对象状态发生变化时，通知组件
  - 比如对象创建、删除、变化等
  - 组件需要和API Server提前建立一个HTTP连接，API Server会通过stream的方式将更新(完整的对象信息)随时发送过来
  - 组件(比如RS的controller)收到更新后，就可以开始进行具体的处理

实际上，API Server 只是提供了一个看板，呈现了对象的信息。而其他组件要求 API Server 在特定对象的状态发生改变的时候通知它们。API Server 会通过流的方式把对象信息发送过来。这其实就是一个 subscribe-publish (订阅-发表) 机制。大家可以订阅自己想关注的 topic，然后当 topic 发表一个东西的时候，通知器就会通知所有订阅的人。

我们有各种各样的 client 建立监听连接，可能是各种各样的内部 controller。假设用户提供 kubectl 创建 pod-xyz，所以 API Server 会先更新 etcd，然后再返回给 client controller。client 拿到 Pod 信息就可以做对应的事情。

## ▶ API Server的监听机制 (2)

- 监听机制案例



## ▶ API Server的监听机制 (3)

- Kubectl同样可以使用监听机制来查看集群中的对象状态变化

```
$ kubectl get pods --watch
```

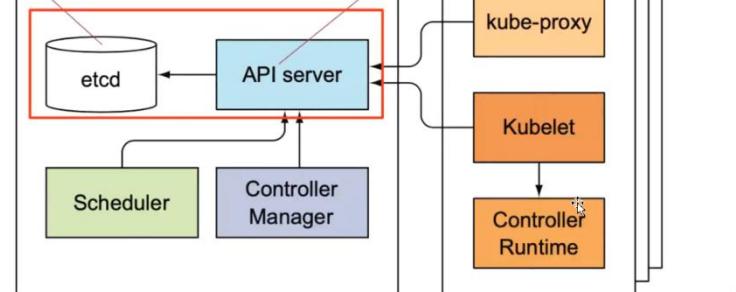
NAME	READY	STATUS	RESTARTS	AGE
kubia-159041347-14j3i	0/1	Pending	0	0s
kubia-159041347-14j3i	0/1	Pending	0	0s
kubia-159041347-14j3i	0/1	ContainerCreating	0	1s
kubia-159041347-14j3i	0/1	Running	0	3s
kubia-159041347-14j3i	1/1	Running	0	5s
kubia-159041347-14j3i	1/1	Terminating	0	9s
kubia-159041347-14j3i	0/1	Terminating	0	17s
kubia-159041347-14j3i	0/1	Terminating	0	17s
kubia-159041347-14j3i	0/1	Terminating	0	17s

其实我们这里就把监听机制讲完了，就是订阅-发表机制。

## ▶ 到目前的内容

存储集群状态，主要是API对象的Spec文件，以及运行时变化的Status等

- 唯一的访问etcd的组件，是其他组件交互的中心。
- 提供操作API对象的CRUD接口
- Authentication→Authorization→Admission
- 监听机制很关键！

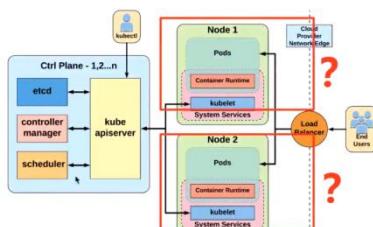


我们现在已经讲完了 etcd 和 API Server。有了监听机制之后，我们不同的模块就可以做协作。我们继续讲 Scheduler 和 Kubelet。

## Scheduler

### 回顾：Pod的创建

- Pod的描述文件中，没有指定在哪个Node跑？



```
apiVersion: v1
kind: Pod
metadata:
  name: pod-example
spec:
  containers:
  - name: nginx
    image: nginx:stable-alpine
  ports:
  - containerPort: 80
```

单个容器的Pod案例

这是一个静态的配置，我们不可能写死了一定要在哪个 Node 上跑。因为 Node 我们可能有新加入和删除的。我们需要把这个关系解耦。所以在真正运行的时候，我们需要一个

Scheduler 来决定 Pod 跑在哪个节点上。这在所有分布式的场景下都是需要的。

## 什么是Scheduler ?

- 简单来看：Scheduler决定一个Pod在哪个Node上跑
- 实际上：
  - Scheduler涉及到大量的策略，需要根据workload的具体要求来进行综合判断，并且尽量达到一个满足条件的，最优的，方案
- Workload的要求通常包括：
  - 硬件需求(比如特定的资源量)
  - 亲和性 (affinity/anti-affinity)
  - 标签匹配等

调度本身就是一门独立的学问，它的策略非常多。讲到调度的时候，我们有很多很多调度策略，这是调度核心所在，要选择策略做到满足条件的最优的方案。亲和性的例子就是绑定 CPU，设定一个 CPU 跑在一个核上。在云环境中，我们可能希望跑在某个特定的 zone 上使得我们访问更快。

## Scheduler的做与不做

- Scheduler做什么？
  - 通过API Server的监听机制，监听Pod的创建
  - 如果新创建的Pod没有指定对应的Node，根据Pod的要求和当前集群的状态，给Pod分配一个Node
  - 将这个Node的信息更新到Pod的配置中，把更新的Pod配置交给 API Server
- Scheduler不做什么？
  - Scheduler不在具体的Node上去创建Pod，启动容器等
  - 这是Kubelet根据Scheduler更新的Pod信息来完成的(课程后续会介绍)

Scheduler 要监听 Pod 的创建，它要给 Pod 指定一个对应的 Node。接下来，我们把 Node 的信息更新到 Pod 的配置中，通过 API Server 写到 etcd 中。所以 Scheduler 本身也不创建 Pod，只是做分配 Node 的事情。创建 Pod 是 Kubelet 去完成的。

## 小明的困惑

Scheduler到底怎么知道应该怎么分配Node给Pod呢？



小明

这是一个很开放的问题，我们来看一个  
K8s中相对基础的实现②

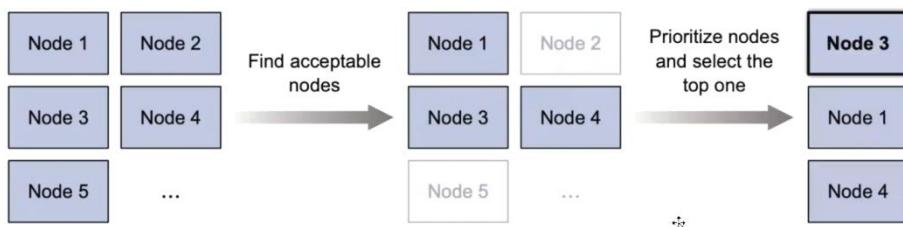


老王

K8s 的实现是两阶段的实现。首先就是通过 label 和 spec 来找到满足条件的 Node。然后从中找出一个最佳的 Node。

## Scheduler的默认实现

- **默认情况下，Scheduler的选择分为两部分**
  - 从一堆Node中，找到满足条件的Node
  - 从满足条件的Node中，选出“最佳”的Node作为最终选择



## Scheduler的默认实现(2)

- **从一堆Node中，找到满足条件的Node (根据大量的条件去筛查)**
  - 这个Node是否满足Pod需要的硬件资源？
  - 这个Node是否已经过载了？比如内存或者硬盘已经超负荷了？
  - 这个Pod是否要求被调度到特定的Node上？（比如满足某个条件的名字）
  - 如果Pod有Node selector选项，当前这个Node的label是否满足要求？
  - 如果Pod要求绑定特定的Node端口，当前这个Node的该端口是否已经被占用了？
  - 如果Pod需要使用特定类型的Volume，这个Volume能否在该Node上被mount上去？
  - 这个Pod和Node之间是否能够通过Taints的要求？
  - Pod是否有亲和性的指定？

初筛都有很多很多条件在里面，找到了满足条件的 Node。

## Scheduler的默认实现(3)

- **从一堆Node中，找到满足条件的Node**
- **从满足条件的Node中，选出“最佳”的Node作为最终选择**
  - 案例：当前集群中，有两个Node可以用来跑Pod，其中一个已经运行了10个Pod，另一个还没有运行任何一个Pod
    - 怎么选择？
  - 在这个场景下，Scheduler会倾向于选第二个Node，可以避免第一个Node上的干扰或者竞争

比如内存带宽、网络带宽都会产生竞争。（考量性能）

- 如果这两个Node是在云上租的节点
  - 在这个场景下，Scheduler也可能倾向于选第一个Node，这样可以将第二个Node释放回去，减少开销

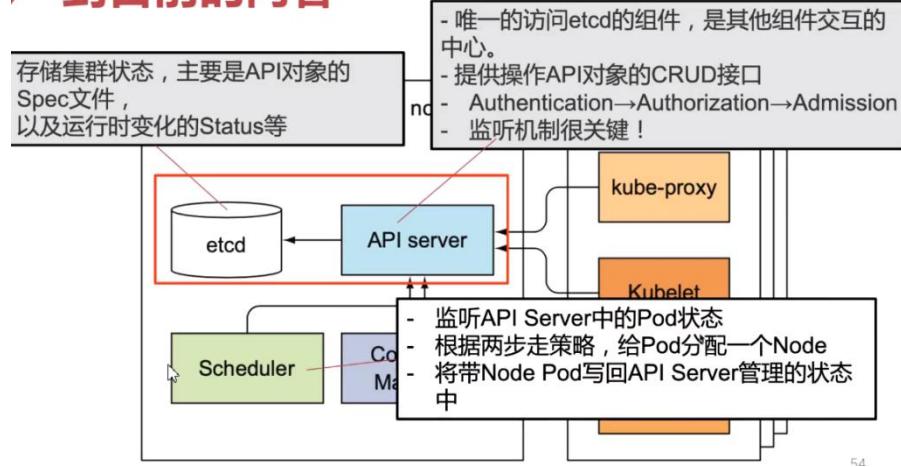
如果云上是弹性付费的情况，那么我们释放第二个可以节约钱。（考虑钱/资源利用率/资源利用效率）。

- 从满足条件的Node中，选出“最佳”的Node作为最终选择
  - 最佳是一个十分看场景的表述
  - 通常来说，集群管理员需要充分考虑集群中的负载，部署的情况，支持的应用模型（如微服务、Serverless）等多种因素，综合考虑“最佳”的实际意义
- 更复杂的考虑：
  - 容错：一个ReplicaSet的多个Pod应该分布在不同的Node
  - ML可以被用来自动生成调度决策，etc.

“最佳”是比较主观的概念，里面涉及到考虑问题的角度。会综合考虑负载、部署、应用模型等方面。一个很好的 Serverless 就是每个请求过来，里面创建做完再删掉。云服务商会提供一些缓存，但是没人用的时候资源就是闲置了。

更复杂的考虑就是需要考虑 Pod 和 Pod 之间的关系（Replica Set 的排他性）。最近在调度中，使用机器学习（ML）的越来越多。这是因为我们要考虑的东西太多了，干脆就使用机器学习来做了，在集群运行过程中，我们可以不断地调整我们的机器学习的模型。

## 到目前的内容



## Kubelet

Kubelet 是分布式环境里每个 Node 里的掌控者。

## Scheduler不负责启动Pod，Kubelet负责！

- 相比其他的K8s核心组件，kubelet是运行在Node上的
- 每个Node一个，是Node的实际管理者
- 负责真正地创建、启动、销毁Pod及内部的容器

Scheduler 只负责分配 Node 给 Pod，而 Kubelet 就需要真正创建这个 Pod。

## Kubelet的工作 (1)

- **初始阶段：**
  - Node上的Kubelet启动后，会连接API Server，告诉API Server这个Node是可用的了(会创建一个Node对象)
  - 之后，Scheduler等其他的组件，就会知道集群中多了一个Node，可以将Pod运行在上面
  - 类似的，如果Node销毁的话，其他组件会看到Node对象消失

## Kubelet的工作 (2)

- **运行阶段：**
  - Kubelet会向API Server注册监听Pod的事件
  - 对于每一个新的Pod，检查它的Node是否是自己(Node是Scheduler分配的)
  - 如果是自己的话，Kubelet会把这个Pod的信息拉到本地
  - 创建Pod要求的容器，配置对应的网络，挂载Volume等等
- **Kubelet还需要向API Server更新Pod信息**
  - Pod是什么状态？创建中，运行中等

每一个新 Pod，Kubelet 就要检查这个 Pod 是不是属于自己的这个 Node。如果属于就要把信息拉到本地，并且创建 Pod。

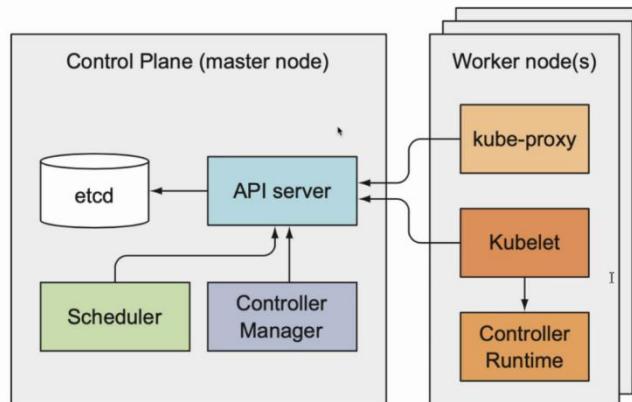
## 下节课：Kubernetes ( 6/完结 )

- 从K8s的内部实现到minik8s ( 下 )
  - K8s中的Controllers是怎么实现ReplicaSet等功能的？
  - K8s插件
- K8s总结
- 其他编排系统案例：Borg设计
- Service Mesh

2022/4/24

开始我们准备的时候，并没有发现有这么多，但是发现只有讲细了才方便写作业。

## 回顾：miniK8s的一个参考架构



因为我们做的是分布式编排系统，内容还是很多的，我们实现的时候没有什么限制。我们先回顾一下上次的参考架构，包含 etcd 存储所有的持久化数据，只有 API Server 会和 etcd 通讯。其他的部件只和 API Server 通讯。我们 API Server 是一个中心化的节点，其他部件通过监控的方式来监控 API Server 进行工作。上节课讲了除 Controller Manager 之外的部件。

## 回顾：K8s中etcd的数据存储

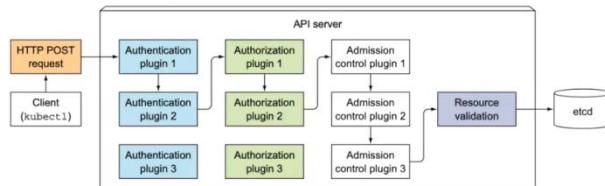
- 放在/registry, 按目录层级方式组织

```
$ etcdctl ls /registry  
/registry/configmaps  
  
$ etcdctl ls /registry/pods  
/registry/pods/default  
  
$ etcdctl ls /registry/pods/default  
/registry/pods/default/kubia-159041347-xk0vc  
/registry/pods/default/kubia-159041347-wt6ga  
  
$ etcdctl get /registry/pods/default/kubia-159041347-wt6ga  
{"kind": "Pod", "apiVersion": "v1", "metadata": {"name": "kubia-159041347-wt6ga",  
"generateName": "kubia-159041347-", "namespace": "default", "selfLink": ...}
```

etcd 有固定的接口和文档，我们只需要使用 etcd 即可。

## 回顾：API Server

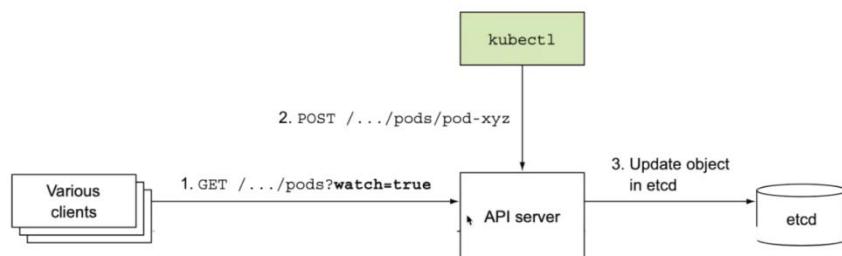
- API Server连接着etcd存储和控制面的各个组织
- API Server的Validation流程
  - Authentication → Authorization → Admission Control



API Server 连接控制面的各个组织。我们还讲了 Validation 流程，但是大家做 Lab 的时候不一定要这样去做。

## 回顾：API Server的监听机制

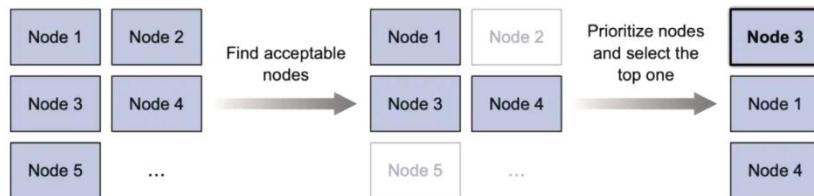
- 允许组件通过watch的机制，监听集群中的状态的变化
- 打通集群中的各个组件的关键机制



# 回顾：K8s中的Scheduler

- 默认情况下，Scheduler的选择分为两部分

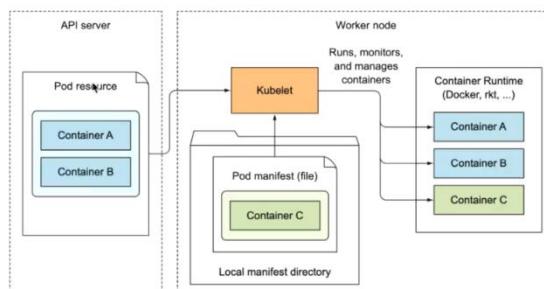
- 从一堆Node中，找到满足条件的Node
- 从满足条件的Node中，选出“最佳”的Node作为最终选择



Scheduler 分为两步，初筛和找最佳。最后的 kubelet 负责单个 Node 的控制，比如创建出相应容器和做相应的任务。

# 回顾：Kubelet的工作

- 负责管理单个Node内部的Pod
- 根据从API Server处获得的Pod信息，执行对应的操作



## 到目前的内容

存储集群状态，主要是API对象的Spec文件，以及运行时变化的Status等

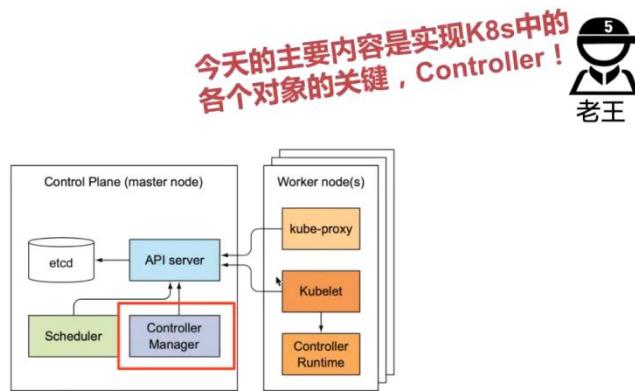
- 唯一的访问etcd的组件，是其他组件交互的中心。  
- 提供操作API对象的CRUD接口  
- Authentication→Authorization→Admission  
- 监听机制很关键！

- 监听API Server中的Pod状态  
- 根据两步走策略，给Pod分配一个Node  
将带Node Pod写回API Server管理的状态中

- 监听Pod信息，检查对应的Node是否归自己管  
启动、运行、检测、销毁对应的Pod实例  
和API Server同步Pod的状态信息

假设我们要实现一个 ReplicaSet，我们的核心就是实现一个 Controller Manager。

# 我们离实现一个ReplicaSet的机制还有多远？

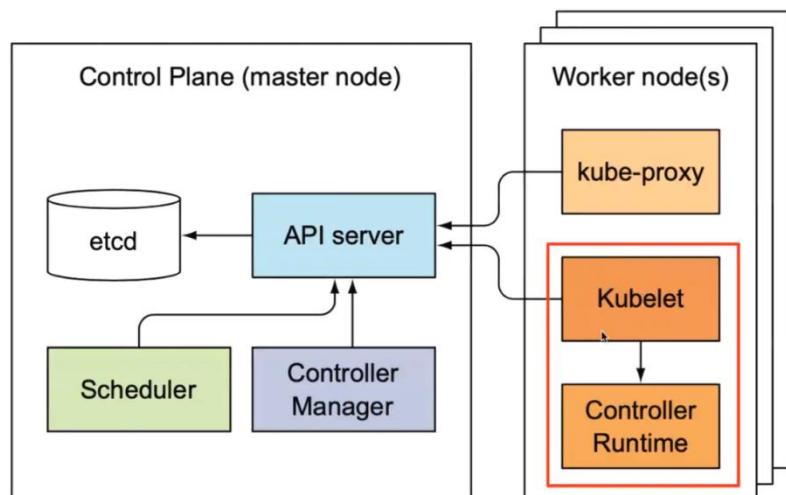


## Kubelet

Q: K8s 是怎么启动容器的？

A: 首先，我们要讲 K8s 是一个比较 high-level 的概念，使用 Pod 和 Service 来管理对象，但是最终我们还是要启动容器的。需要通过 CRI 连接连到容器上去。

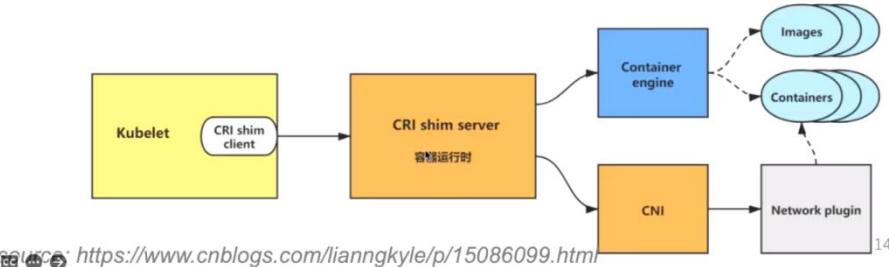
## Kubelet到容器运行时



最终，kubelet 是需要连到 controller runtime (container runtime)。

## Kubelet到容器运行时

- Kubelet 通过K8s的CRI (Container Runtime Interface, 容器运行时接口) , 来访问底层的具体容器系统
- 实现了CRI接口的容器运行时 : CRI Shim (gRPC server)
- Shim通过unix socket来处理kubelet的调用

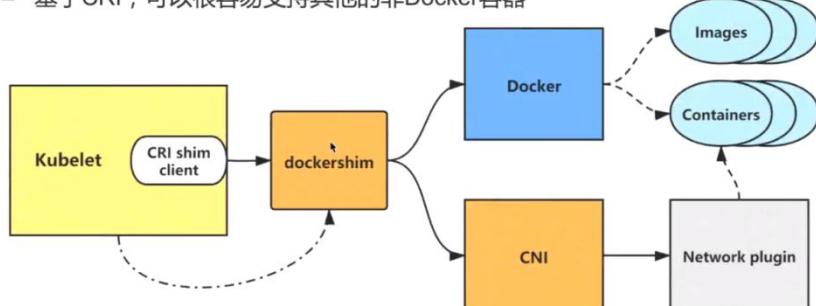


其实 K8s 里就做了 CRI 接口的容器运行时，我们可以 kubelet 接下来的东西都是和容器运行时相关的内容。CRI shim server 是轻量级的，可能只做了一些安全检查，最终我们掉到 Container Engine 里去。最新的 K8s 可能直接通过 runc 去创建容器，就不走 docker 包装的接口了。

Kubelet 其实也是实现了一个 Shim Server。Shim Server(dockershim)其实就是监听 kubelet 的 K8s 的对容器操作的请求，转化成 REST API 发送给 docker。

- Kubelet和CRI怎么对接到Docker呢？

- Dockershim: dockershim 监听 kubelet 的请求，将其转化为 REST API，发给 docker
- 基于 CRI，可以很容易支持其他的非 Docker 容器



因为 CRI 本身就是一个兼容性很高的接口，所以我们对不同的后端兼容很容易就把 docker 换成 runc、katacontainer 等容器运行时。

CRI 是比较复杂的，实现了很多接口。蓝色的是和 Pod 相关的、橙色的是和容器相关的。还有一些别的操作方法。

## Kubelet到容器运行时 (3)

- **CRI**
  - 提供了Pod、容器等的生命周期管理、数据交互等接口
- **MiniK8s**
  - **Option-1:** 自己定义miniK8s和具体的容器运行时(如docker)之间的协议，来实现Pod和容器的生命周期管理和交互
  - **Option-2:** 基于CRI接口去设计和实现miniK8s中的kubelet，直接复用现有的软件栈(不需要支持全部的CRI接口)

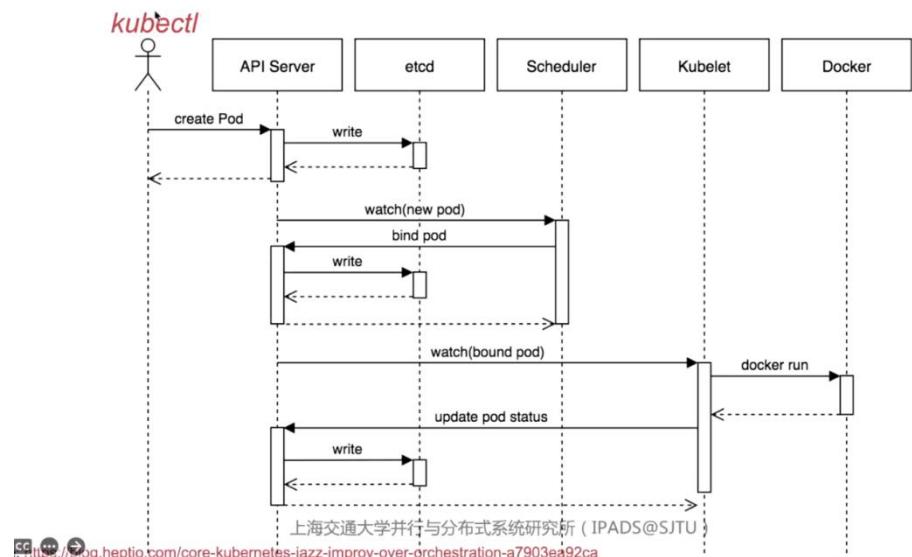
RuntimeServiceServer		
RunPodSandbox	CreateContainer	Version
StopPodSandbox	StartContainer	Status
RemovePodSandbox	StopContainer	UpdateRuntimeConfig
PodSandboxStatus	RemoveContainer	ExecSync
ListPodSandbox	ListContainers	Exec
ListContainerStats	ContainerStatus	Attach
ReopenContainerLog	UpdateContainerResources	PortForward

我们不要求大家做一个兼容 CRI 的接口。我们只要实现课程要求的文档上的内容就行了。我们给大家两个方案，第一种就是自己实现和 docker 之间交互的协议，第二种就是根据 CRI 接口去实现 miniK8s 中的 kubelet，这样我们可以复用一些 docker CRI 相关的接口。

## 启动一个 Pod 的工作流

我们使用 kubectl 去控制这么一个系统（创建 Pod）。现在学完了各个部件之后，对这个工作流有新的认识了。

## 基于API Server的一个完整的工作流



流程如下：

1. kubectl 给 API Server 发出创建 Pod 的请求。
2. API Server 和 etcd 交互，更新 Pod 的信息，并且返回给 kubectl 操作完成。
3. 调度器从 API Server 这里监测到 Pod 信息的变化，给 Pod 绑定了一个 Node，并更新。
4. API Server 在 etcd 中更新 Pod 信息。并且通知调度器操作完成。
5. kubelet 监听到 Pod 的信息，开始在本地通过 Docker 启动 Pod，并告知 API Server。
6. API Server 在 etcd 中更新 Pod 的运行信息和状态。

这个时候我们就完成创建了 Pod, API Server 完成了其中的三个步骤。大家实现出来可能也是一个类似的效果。通过向 API Server 发送请求，并且 API Server 根据关注来推送信息。

这里就讲完了 kubelet 的功能和接口。

## Controller

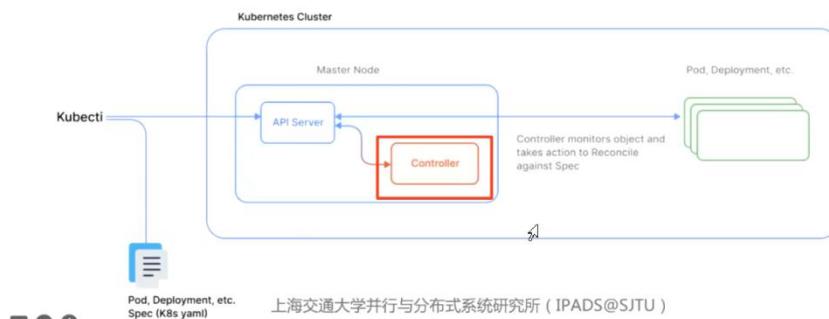
# K8s的Controller

- Kubernetes 通过运行一组 **Controllers**, 来实现集群中的真实状态和期望状态
- 比如：
  - Replication Manager/ReplicaSet Controller: 管理和维护集群中的 RS 等 replica 相关的对象
  - Node Controller: 管理集群中的 Node 节点的状态，并且在 Node 节点加入和退出时动态更新
- 基本上，每一类 K8s 对象，都会有自己的 controller 来对其进行管理

RM/RC 的期望值就是 replica 的数量，实际值就是现在 replica 的数量。Controller 就是保证实际状态等于期望状态。

# K8s的Controller

- 抛开具体的任务而言，Controller 会监听 K8s 对象的状态，然后针对状态进行操作
  - 监听：① 通过 API Server 的 watch 机制；② 主动去 query 信息



Controller 和我们之前讲的对象没什么区别，都是视同 watch 机制和 query 机制从 API Server 中得到信息。Controller 有一段时间没有收到 API Server 的信息、或者需要一个对象的额外信息的时候，都需要主动向 API Server 询问。

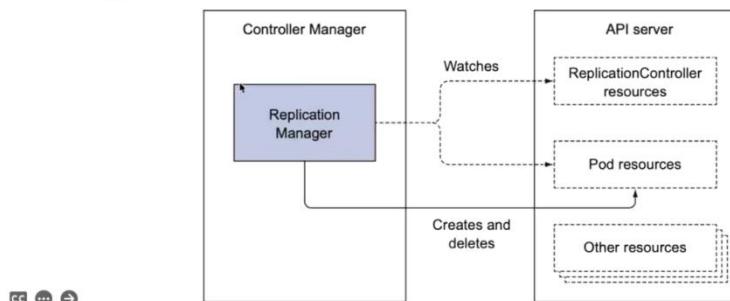
## 回顾 : Replication Controller (RC)

- RC对应的功能 : 保证集群中指定Pod副本数量
  - 多个或者1个
- 两种处理方式 :
  - 当对应的Pod数量少于指定数目时 , RC自动启动新的Pod副本
  - 当对应的Pod数量多于指定数目时 , RC自动杀死多余的副本
- 场景 : 当通过RC运行指定的Pod , 并配置为1时
  - 即使出现错误导致Pod退出 , RC也会立即重启一个Pod实例
  - 高可用
- RC是K8s中 , 早期引入的保证Pod高可用的API对象

通过这个方法 , 用户不需要主动调整 Replica 的数量。RC 其实对应了 Controller Manager 中的 Replication Manager。

## Replication Manager管理RC

- Replication Manager负责管理RC的生命周期
  - 监听两类资源 : **ReplicationController(RC)** 和 **Pod**
  - 根据RC对Pod的期望和真实Pod的状态 , 来增加或减少Pod的实例



它监听 ReplicationControllerResource 和 Pod resources (放了 Pod 实例信息: 有哪些 Pod 已经创建出来了)。它核心就是监控这两个, 根据 Replication 对 Pod 的期望增加或减少 Pod 的实例。

# ReplicaSet/Deployment的Controller

- **ReplicaSet Controller**

- ReplicaSet controller的任务和Replication Manager几乎一模一样
- 值得注意的是，ReplicaSet和Replication都不会手动去运行/暂停Pod
- 它们只更新Pod的spec，来增加或停止Pod (平台具体怎么做不关心)

- **Deployment Controller**

- 根据Deployment对象描述的state，来更新集群中的状态
- 比如，滚动升级的方式: **maxSurge** 和 **maxUnavailable** 参数

它做的事情和 Replication Controller 差不多，更新 Pod 的 spec 来增加或者停止 Pod。ReplicaSet 是一个 API Server 中的对象，它只是把自己的 spec 放在 API Server 里面。Replication Manager 读到了这个 Spec 中，知道了对某个 Pod 应该维护住多少的 replica。

## Node Controller

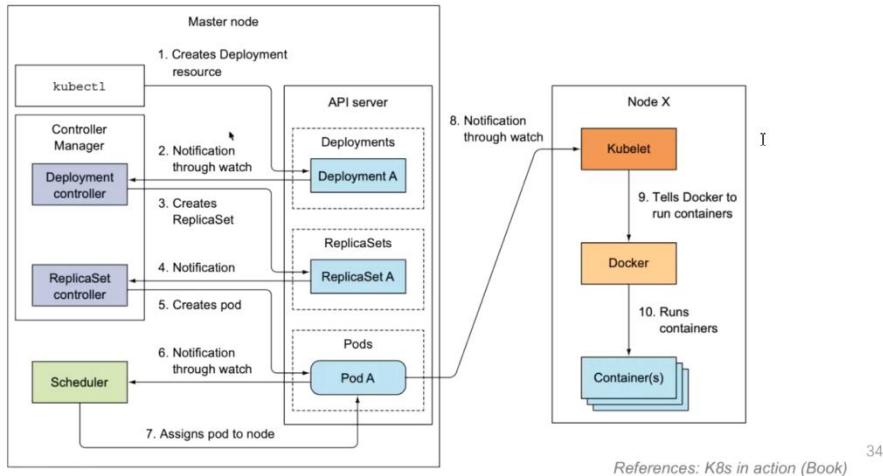
这个我们之前讲的比较少，因为 Node 对象涉及到物理节点。和别的对象关系不是很大。我们要做多机 miniK8s，可以实现这个 Node Controller。它要保证 Node 对象和运行的 Node 是匹配的。新的节点的信息要同步到 API Server 里面，这样我们的 Scheduler 才知道集群加入了一个新的 Node，下次调度 Pod 的时候可以纳入考虑。

## Node Controller

- K8s中，通过Node资源(对象) 来记录当前集群中的工作Node
- Node controller需要保证Node对象和当前真实的运行的Node是匹配的
- 实时更新Node的健康信息等
- Scheduler等使用的Node信息来自于Node controller的维护和更新
- 支持多Node的一个关键部分

# 创建一个 Deployment 的工作流

## 完整流程与Controller间协作: 创建一个Deployment



根据这个例子来看，它的流程一样很复杂。

1. kubectl 给 API Server 发起创建 Deployment 的一个请求。API Server 创建好对应的对象，并且保存到 etcd 中。
2. Deployment Controller 通过 watch 机制，从 API Server 处监听到 Deployment 这类对象的新增，开始处理。
3. Deployment Controller 根据 Deployment 中的说明，创建了一个新的 ReplicaSet 对象更新到 API Server 中。API Server 创建了对应的 ReplicaSet 对象并且保存到 etcd 中。
4. API Server 把 ReplicaSet 对象的新增通知到 ReplicaSet Controller 中。这是因为 ReplicaSet Controller 监听了所有 ReplicaSet 对象的操作的处理。
5. ReplicaSet Controller 发现自己需要更多的 Pod 来满足 ReplicaSet 的要求，于是创建了 Pod 对象，并且由 API Server 保存到 etcd 中。
6. 新创建的 Pod 的这个时间又会通知到 Scheduler，因为 Scheduler 会监听所有 Pod 的创建。
7. Scheduler 根据当前集群状态和自己的策略，给 Pod 分配了一个 Node，并且更新回 API Server 中。
- 8-10. Pod 的更新会同时通知到 Kubelet，一旦其发现自己需要负责这个 Pod，就会开始调用 Docker 等来启动容器。

现在我们创建的一个 high-level 的 Deployment，它里面套了 ReplicaSet 和 Pod，涉及到两个 Controller 的启动。因为我们的信息基本上都是 API Server 推送过来的，比如说我们现在要开始滚动升级，先关掉了一个 Pod。此时 Deployment Controller 和 ReplicaSet Controller 都会收到一个 Pod 被关掉的消息推送。那么怎么判断此时谁应该来负责这件事情呢？如果两个 Controller 并行地启动 Pod，那么我们最终就会启动两个 Pod。K8s 的做法是让 Controller 自己处理，需要让 Controller 自己判断什么时候是 Pod 因为滚动更新被关闭了。

## Controller的核心组成

- Controller在一定程度上，可以理解为是运行在控制面的一个API Server的client (所有的组件都可以这么理解)
- K8s提供了一系列的库来支持Controller的实现
  - client-go: <https://github.com/kubernetes/client-go>

Controller 其实也是 watch API Server 来进行操作。GO 语言写的可读性也不是能很好，并不建议做 Lab 时很仔细地读大型项目的代码。我们更多是希望讲一些语言无关的怎么实现的内容。

- 一个Controller通常包含两个主要的组件：
  - **Informer/SharedInformer**
  - **Workqueue**
  - ↑ 这两个组件都由client-go库提供

### Informer

## Controller的核心组成(2): Informer

- **Informer**
  - Controller需要和API Server之间不停地同步关注的一些Object的信息，来判断是否要执行操作
  - 可能会导致大量的数据传输 (和集群中的对象数量正相关)
  - **Informer:** 将Object的数据缓存在controller本地，只监听更新并及时同步到本地cache中
  - **不足：**有可能存在多个controller监听同一个对象 → 存在多份缓存，并且缓存之间状态同步会较为复杂，甚至出现状态分叉
    - → SharedInformer

Informer 可以认为是一个缓存，比如我们的 Controller 老是要和 API Server 同步一些关心的 Pod 的信息。这个过程里，我们更新的是 Pod 的状态，Object 本身的数据，API Server 不再更新给 Controller，为了避免每次都要往 API Server 去要，所以可以在本地做缓存。

如果我们两个服务器上有两个不同的缓存，如果状态修改了，可能就需要保证缓存的一致性。缓存管理会变的很复杂，所以就需要 SharedInformer，共享一个缓存。

## Controller的核心组成 (3): SharedInformer

- **SharedInformer**
  - 同样在Controller侧维护缓存
  - 但是缓存的数据是被多个Controller共享的
  - 避免了多份缓存，以及状态不一致的问题

## › Controller的核心组成 (4)

- 一个Controller包含两个主要的组件：
  - Informer/SharedInformer
  - Workqueue
- **Informer/SharedInformer**

来看一眼代码吧



```
store, controller := cache.NewInformer {  
    &cache.ListWatch{},  
    &v1.Pod{},  
    resyncPeriod,  
    cache.ResourceEventHandlerFuncs{},
```

### Controller

- 一个Controller包含两个主要的组件：
  - Informer/SharedInformer
  - Workqueue
- **Informer/SharedInformer**

```
cache.ListWatch {  
    listFunc := func(options metav1.ListOptions) (runtime.Object, error) {  
        return client.Get().  
            Namespace(namespace).  
            Resource(resource).  
            VersionedParams(&options, metav1.ParameterCodec).  
            FieldsSelectorParam(fieldSelector).  
            Do().  
            Get()  
    }  
    watchFunc := func(options metav1.ListOptions) (watch.Interface, error) {  
        options.Watch = true  
        return client.Get().  
            Namespace(namespace).  
            Resource(resource).  
            VersionedParams(&options, metav1.ParameterCodec).  
            FieldsSelectorParam(fieldSelector).  
            Watch()  
    }  
}
```

```
store, controller := cache.NewInformer {  
    &cache.ListWatch{},  
    &v1.Pod{},  
    resyncPeriod,  
    cache.ResourceEventHandlerFuncs{},
```

watch 就是被动地去看，list 就是主动的 query。在 function 里面，就是指定了我们要关注 resource 和 namespace。

## Controller的核心组成 (4)

- 一个Controller包含两个主要的组件：

- Informer/SharedInformer

- Workqueue

- Informer/SharedInformer

```
lw := cache.NewListWatchFromClient(  
    client,  
    &v1.Pod{},  
    api.NamespaceAll,  
    fieldSelector)
```

代码吧



老王

SharedInformer基于lw来构造，其和Informer是类似的

```
lw := cache.NewListWatchFromClient(...)  
sharedInformer := cache.NewSharedInformer(lw, &api.Pod{}, resyncPeriod)
```

## Workqueue

SharedInformer 下面就有一个 Workqueue，给每个 Controller 提供队列支持。

## Controller的核心组成 (5): Workqueue

- Workqueue

- SharedInformer构建了一个Cache，来维护其中的信息的变化
- 需要有独立的队列系统来维护每个Controller需要处理的任务
- Workqueue: 当一个对象的状态发生变化时，SharedInformer会通过事件handler来将一个key放在Controller的workqueue中
- Controller可以通过启动工作线程，来在workqueue中获取自己需要处理的对象的key，开始操作

## Controller的核心组成 (5): Workqueue

- Workqueue

- 基于client-go的库创建workqueue

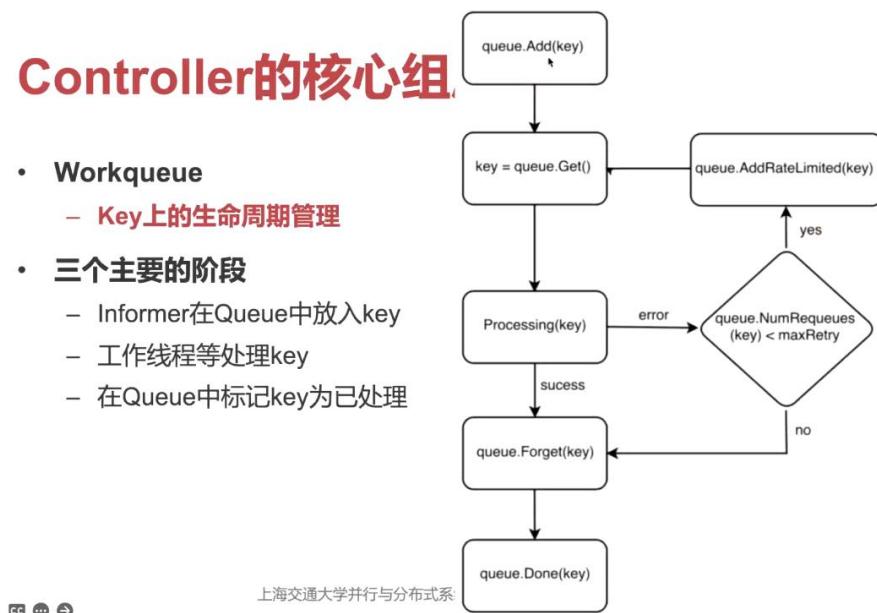
来看一眼代码吧



```
queue :=  
workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter())
```

## Controller的核心组件

- **Workqueue**
  - Key上的生命周期管理
- **三个主要的阶段**
  - Informer在Queue中放入key
  - 工作线程等处理key
  - 在Queue中标记key为已处理



## 案例分析：Kubewatch

### • 功能描述：

- Kubewatch会监控K8s集群中的Pod的状态
- 然后将Pod的状态发送到Slack等“社交”平台上
- 基于Golang实现
- 基于K8s client library和K8s API server交互
- 用一个Slack client library和Slack交互

Source: <https://github.com/vmware-archive/kubewatch>

## Kubewatch的说明

### • 核心数据结构(Golang):

```
// Controller object
type Controller struct {
    logger      *logrus.Entry
    clientset   kubernetes.Interface
    queue       workqueue.RateLimitingInterface
    informer    cache.SharedIndexInformer
    eventHandler handlers.Handler
}
```

Controller的log  
和K8s API server交互的接口  
★SharedInformer和Workqueue  
具体的处理函数 (这里和Slack交互)

clientset 就是和 API Server 交互的接口。

## Kubewatch的说明 ( 2 )

- 构建WorkerQueue和SharedInformer:

```
queue := workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter())
informer := cache.NewSharedIndexInformer(
    &cache.ListWatch{
        ListFunc: func(options meta_v1.ListOptions) (runtime.Object, error) {
            return client.CoreV1().Pods(meta_v1.NamespaceAll).List(options)
        },
        WatchFunc: func(options meta_v1.ListOptions) (watch.Interface, error) {
            return client.CoreV1().Pods(meta_v1.NamespaceAll).Watch(options)
        },
    },
    &api_v1.Pod{},
    0, //Skip resync
    cache.Indexers{},
)
```

## Kubewatch的说明 ( 3 )

- 运行Controller:

```
// Run will start the controller.
// StopCh channel is used to send interrupt signal to stop it.
func (c *Controller) Run(stopCh <-chan struct{}) {
    // don't let panics crash the process
    defer utilruntime.HandleCrash()
    // make sure the work queue is shutdown which will trigger workers to end
    defer c.queue.ShutDown()

    c.logger.Info("Starting kubewatch controller")
    go c.informer.Run(stopCh) // Informer开始run !
    // wait for the caches to synchronize before starting the worker
    if !cache.WaitForCacheSync(stopCh, c.hasSynced) {
        utilruntime.HandleError(fmt.Errorf("Timed out waiting for caches to sync"))
        return
    }

    c.logger.Info("Kubewatch controller synced and ready")

    // runWorker will loop until "something bad" happens. The .Until will
    // then refresh the worker after one second
    wait.Until(c.runWorker, time.Second, stopCh)
}
```

启动worker来处理queue上的key对应的对象变化

Source: <https://github.com/vmware-archive/kubewatch>

Worker 把数据往 Slack 上推送。它就是 for 循环处理所有的 item。

## Kubewatch的说明 ( 3 )

- 运行Controller:

```
func (c *Controller) runWorker() {
    // processNextWorkItem will automatically wait until there's work available
    for c.processNextItem() { // continue looping
    }
}
```

```

// processNextWorkItem deals with one key off the queue. It returns false
// when it's time to quit.
func (c *Controller) processNextItem() bool {
    // pull the next work item from queue. It should be a key we use to lookup
    // something in a cache
    key, quit := c.queue.Get()
    if quit {
        return false
    }

    // you always have to indicate to the queue that you've completed a piece of
    // work
    defer c.queue.Done(key)

    // do your work on the key.
    err := c.processItem(key.(string)) // 调用真正的handler

    if err == nil {
        // No error, tell the queue to stop tracking history
        c.queue.Forget(key)
    } else if c.queue.NumRequeues(key) < maxRetries {
        c.logger.Errorf("Error processing %s (will retry): %v", key, err)
        // requeue the item to work on later
        c.queue.AddRateLimited(key)
    } else {
        // err != nil and too many retries
        c.logger.Errorf("Error processing %s (giving up): %v", key, err)
        c.queue.Forget(key)
        utilruntime.HandleError(err)
    }
}

return true
}

```

## ► Kubewatch的说明 ( 5 )

- 来看看效果吧

```

$ wget https://github.com/skippbox/kubewatch/releases/download/v0.0.3/kubewatch.yaml

//Note: update slack channel and slack token values at the configmap object.
//Then deploy kubewatch

$ kubectl create -f kubewatch.yaml

$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
kubewatch  1/1      Running   0          2m

```

Ref: <https://docs.bitnami.com/tutorials/kubewatch-an-example-of-kubernetes-custom-controller>

Source: <https://github.com/vmware-archive/kubewatch>

67

```

skippbox-kubewatch APP 1:04 PM
kubewatch
A pod in namespace kubeless has been created: kafka-0

kubewatch
A pod in namespace kubeless has been created: kubeless-controller-1399132201-wq3j1

kubewatch
A pod in namespace kubeless has been created: zoo-0

kubewatch
A pod in namespace default has been created: get-python-2137859707-3x9wn

kubewatch
A pod in namespace kube-system has been created: kube-addon-manager-minikube

```

## 怎么去快速查看现有K8s controller的实现？

- **代码的位置在：**

- <https://github.com/kubernetes/kubernetes/blob/master/pkg/controller>

- **确定Controller的资源监听范围**

- Controller的构造函数中通常会去创建一个  
Informer/SharedInformer等变种，通过它们的声明去确定资源  
范围

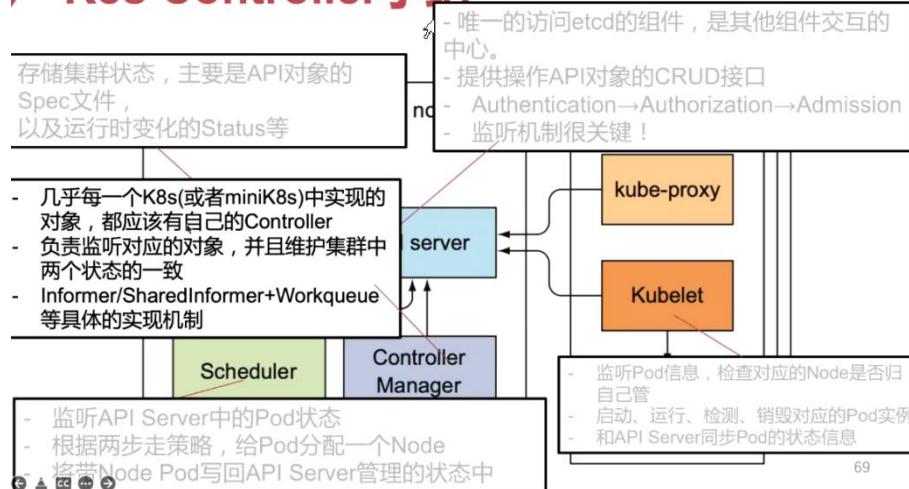
- **确定Controller的处理逻辑**

- 当监测到变化时，做什么操作？
- 重点关注worker(), syncHandler() 等类似的接口函数

一些建议



## ► K8s Controller小结



Controller 也是一些 Pod 来做管理我们所有的 API 对象。具体的实现机制就是 Informer 和 Workqueue。

## Lab 相关

Pod 的话，只要打通 Pod 到 contrainer 的启动，基本上就总体成功了。Service 的话更多是要支持 IP 和支持两个 Pod 之间的通讯。Replica Set 中，如果我们要实现滚动更新，那么我们就要实现了一个 Deployment。DNS 是网络的事情，可以不用去看一个真的 DNS 实现的，它无非就是给你一个名字返回一个 IP。容错其实也是 Replica Set 和 etcd 就可以保证容器面和控制面的 crash 即可。多机就是 Node 抽象以及怎么注册。微服务这边就需要讲 Service Mesh。Serverless 的应用可能比微服务还要简单，更多是在 Function 抽象上。

**2022/4/26**

## 扩展 K8s: CRD 和 K8s 插件

### CRD (CustomResourceDefinitions)

- **K8s通过期望状态，将controller的任务做了很干净的划分**
  - RS controller只保证Pod的数量和期望一直，不启动Pod
  - Kubelet只启动和管理分配给自己节点的Pod
- **但是，实际状态 == 期望状态**
  - 这里的两个状态有很强的语义在里面
  - 以RS为例: pod个数等于RS中的replica数要求RS的API对象中，有replica数的这个指标
  - **需求**: 如果，现在我们希望一个ReplicaSet的期望状态是一个区间，比如真实的Pod数应该在min\_replica 和 max\_replica之间怎么办？

我们整个过程中都在强调 K8s 管理状态的方式就是让实际状态和期望状态等同，不同的 Controller 管不同的事情。因为状态这个东西有清晰的语义在里面，比如扩增/删除我们的 Replica 数量，这些语义都是需要用户定义的。所以对于不同的状态我们都有类似的实现。比如滚动更新的时候实例数量应该在什么范围内，总的来说我们的任务划分比较清晰，用户只需要提出需求，Controller 去实现。

坏处就是需求比较固定，比如我们现在 ReplicaSet 不希望 replica 是固定值，那么怎么做呢？就有了 CRD 这个抽象，说白了可以自定义对象类型和 Controller 类型。

## CRD (CustomResourceDefinitions) (2)

- 由于RS本身的spec就没有min\_replica , max\_replica的值，实现这点其实比较困难
- K8s提供了CRD的抽象：
  - 允许用户自定义类型
  - 用户可以自定义controller
  - 通过CRD可以实现一个新的ReplicaSet，并且几乎允许开发者在K8s做任意的扩展

这样我们就可以基于 CRD 模型实现一个新的 ReplicaSet，这样我们就可以做相应的扩展。所以 K8s 的扩展自由度是比较高的。

下面就有一个例子：

我们可以看到 kind 字段里是“CustomResourceDefinition”，也就是 CRD，这个类型名字为“Crontab”表示这是一个新的对象。定义好对象类型之后，我们就可以设置对应类型的对象实例。

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name must match the spec fields below, and be in the form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage version.
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                  image:
                    type: string
                  replicas:
                    type: integer
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # singular name to be used as an alias on the CLI and for display
    singular: crontab
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: CronTab
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
      - ct
```

### 新的类型对应的对象实例

Source: <https://kubernetes.io/docs/tasks/extend-kubernetes/> 上海交通大学并行与

### 新的对象类型

接下来是 K8s 中的插件，比如图形化功能不够丰富，我们可以自己写一个 dashboard 插件。插件其实就是一些特殊的 Pod。

## 插件 ( Add-on)

- 插件是另外一类扩展K8s能力的方式
- 和大家平时给IDE等加插件的方式一样，k8s也允许通过插件给集群注入新的一些能力
  - 比如DNS lookup、将多个HTTP service暴露成一个IP、K8s的dashboard等等
- 从系统角度来看，插件其实就是一些特殊的Pod
  - 通过YAML等部署到集群中
  - 以Deployment、ReplicationController(或RS)、DaemonSet等形式部署

kube-system 已经内置了一些特殊的 Pod（插件）。这些插件用户就不需要实现它了。

## 插件 ( Add-on) (2)

```
$ kubectl get rc -n kube-system
NAME                      DESIRED   CURRENT   READY   AGE
default-http-backend       1         1         1       6d
kubernetes-dashboard      1         1         1       6d
nginx-ingress-controller  1         1         1       6d

$ kubectl get deploy -n kube-system
NAME        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
kube-dns    1         1         1           1          6d
```

讲到这里，K8s 基本上讲完了。后面的内容 K8s 还会不停地出现，比如我们讲网络的时候。网络本身是云 OS 中主要的一部分，在做 K8s 的时候我们也要用到网络。Service 需要通过 IP 来找到 Pod，这就是网络的一种应用。

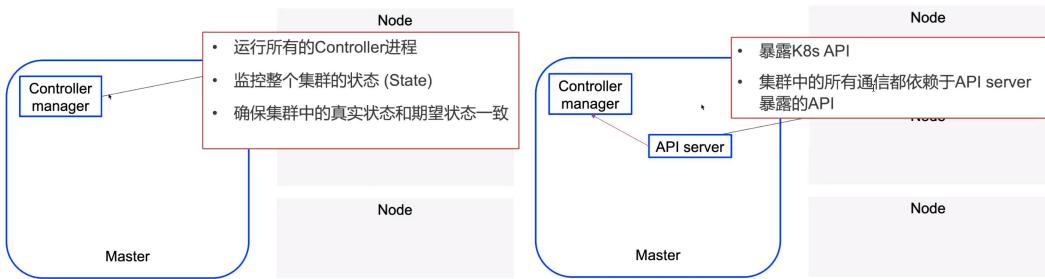
## Outline : K8s总结+云原生网络

- K8s总结
- Borg系统分析
- 云原生网络
  - API gateway
  - Service mesh

## K8s 总结

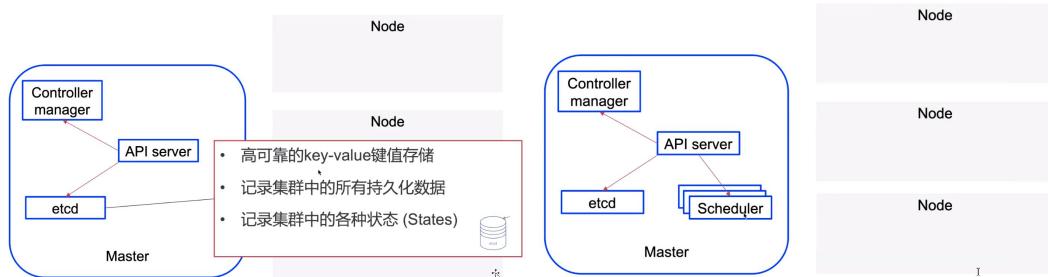
最开始的时候我们也讲过 K8s 的架构，现在再听会有不一样的感觉。首先是经典的主从架构，分为 master 和 worker node。

## K8s组件: Master



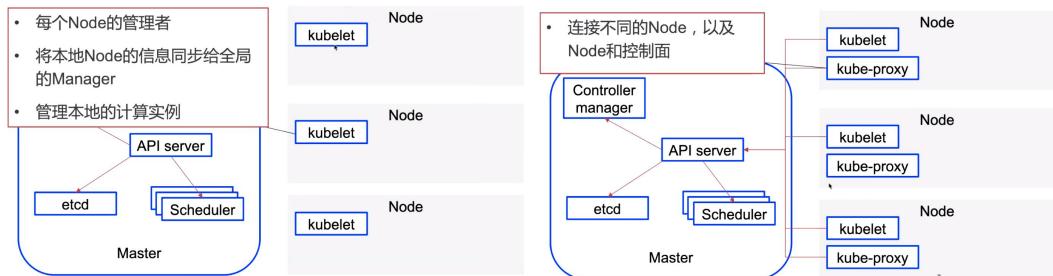
API Server 通过 watch 的方式把信息转发给关注的人。

## K8s组件: Master



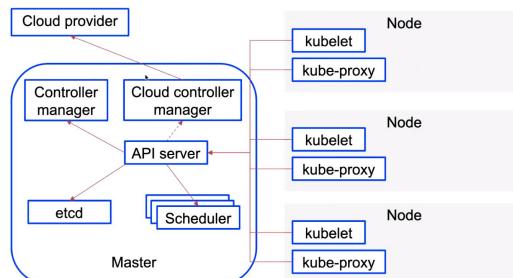
Scheduler 负责资源调度，当我们想新创建一个 Pod 的时候，scheduler 决定这个 Pod 去哪，但不执行具体的创建 Pod 的行为。

## K8s组件: Node

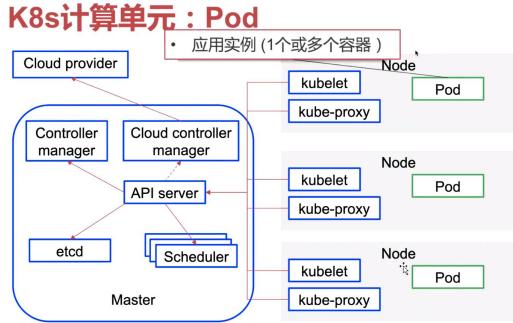


Kubelet 管理自己这个 Node 的信息，当 Scheduler 告诉它要创建 Pod 的时候，它负责去部署这个 Pod。

kube-proxy 是网络相关的。Node 和 Node 之间都是使用 kube-proxy 相连来配置网络，比如让不同的 Service 相连、让不同的 Pod 相连。



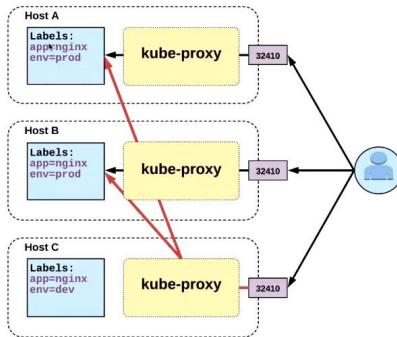
这里还有一个最近没有讲的点就是 Cloud Controller Manager，可以认为它是 Controller Manager 的一部分。同学们的做法就是组几台机器在内网/数据中心网络里做。但是我们实际的云环境中，云业务可能需要不同的云服务提供商的协同，比如某某云 GPU 比较便宜，某某云的 Serverless 比较强，在这种情况下我们就需要做跨云的业务。Cloud Controller Manager 就是连接云服务提供商，它来对接具体的云服务。这就是本地云集群和公有云连接的接口。



Pod就是基本的计算单元，里面包含多个容器共同完成一件事情。Pod是我们要实现的第一个功能。在实现的过程中，我们可以把相应的模块搭一搭。

## Service

- **主要有四种常见的Service**
  - ClusterIP
  - NodePort
  - LoadBalancer
  - ExternalName



主要有四种不同的Service。ExternalName就是用来连接其他服务的。我们的Lab中主要完成ClusterIP的设计。在我们后面做到网络/微服务的时候，我们可能要花更多时间实现外部网络的支持上。

接下来我们要实现两种对象，有Replica这个期望值。Deployment这部分比较复杂，有版本升级的要求。

## Workload之ReplicaSet

- 前面介绍过的ReplicationController(RC)的升级版
- 非常常用的管理Pod备份实例和生命周期的对象
  - 管理Pod的调度、扩容(scaling)、删除等
- 核心任务：始终保证期望数量(Desired)的Pod实例在运行

## Workload之Deployment

- 在ReplicaSet的基础上，进一步封装的workload对象
- 通过声明式(Declarative)的方式来管理Pod
- 提供了回滚(rollback)机制和**版本升级控制**
- 版本升级控制通过: pod-template-hash 的标签控制
- 每一次升级迭代，会给对应的ReplicaSet和Pod打上新的标签



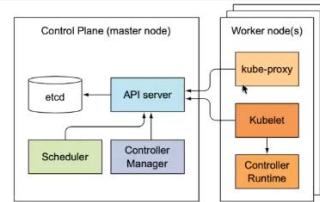
Kubelet的目标就是给你一个YAML，拿到以后去启动一个容器。我们可以使用定义好的CRI协议外接一个CRI兼容的模块去启动这个协议。或者我们后面接一个docker，直接给docker发送即可。我们把这个组织成一个命令发给docker去做相应的操作。

API Server和etcd主要就是提供spec的定义，我们也有yaml的要求，并且实现相应的对象。在此基础之上，我们要提供API对象的管理，还有一个关键的watch的机制。提供watch，我们要让关注的人知道我们的对象发生了变化。其他的过程可以比较简略，比如验证这种概念。

Scheduler这边的核心就是调度的策略。其实实现完watch机制和API对象的管理接口之后，我们无非就是往上搭积木即可。Scheduler无非就是watch API对象Pod的变化，一旦API Server发现有Pod对象的变化，我们就发给scheduler即可。

在我们有 Scheduler 之后，我们就可以做到在多台机器上创建 Pod 了，加上我们之前做的 kubelet 的创建，多机创建 Pod 这个流程已经走通了。再接下来，我们再实现一个个复杂对象。我们在实现的时候可以一个模块一个进程。

## 总结：K8s→miniK8s



- **Kubelet:**
  - 实现Pod的抽象和生命周期管理
  - 关键：怎么和容器运行时引擎交互？（现有CRI或自定义协议）
- **API server+etcd**
  - 实现对集群状态的管理: spec的定义、etcd中的数据存放格式
  - 实现CRUD接口+watch通知机制，验证等过程可以简略
- **Scheduler:**
  - 监听Pod的状态，分配Node，可以有特定的设计的调度策略，也可以选择一些简单的策略，比如Round-robin
- **Controller:**
  - 实现复杂的对象的关键，RS/Deployment
  - 注意分工，已经对一些corner case的处理

## Google BORG: K8s 的前身

这边简单总结了我们的 K8s，我们来看看 K8s 的前身。今天的 K8s 已经是一个比较成熟的系统，但是刚开始还是有一些值得探索的设计。比如 Borg (Eurosyst'15) 就是 K8s 前身的一个系统。K8s 在实现上受了很多 Borg 的影响。当时谷歌已经在美国有很多很大的集群了，因为集群是很贵的，如果机器坏了一片，如果维修修不了了，我们还需要用卡车拉走旧设备拉来新设备。一个数据中心变成了一个需要复杂管理架构的机构了，那么很难使用人工的方法去管理我们的机器，所以需要使用一个管理系统来自动化管理。

关键技术就是：准入限制、超售（比如我们在 XX 云上购买了一个内存，它一般会宣称 1G 是物理内存，而不是虚拟内存。而在公司内部，我们为了让集群利用率升高，需要使用超售的手段）。

## Borg

- Borg是Google内部的集群管理系统
- 能够实现极高的集群利用率，关键技术：
  - Admission control
  - Efficient task-packing
  - Over-commitment
  - Machine sharing

Borg 是用户是运维人员和开发人员，可以认为是一个私有云。当时的负载种类不是很多，但是差异巨大。Borg 主要关注长期运行的服务（Gmail, Google doc 这种要即时做出反应的服务）和批处理服务（时间较长）。这两种任务有比较大的区别。

## 什么是Borg？用户视角的核心概念

- **Borg的用户：**主要是Google的开发者和系统维护者
  - 集群使用者，这是和公有云的一个不同
- **负载类型：**
  - **Production:** 长期运行的服务，主要的场景是对处理时延敏感的请求 ( $\mu\text{s}$  到数百毫秒这个量级)
    - 案例：Gmail, Google doc
  - **Batch:** 单个任务处理较长的复杂（数秒到几天）
    - 对于临时的性能抖动不敏感

K8s 其实我们没有讨论负载是什么样子的。我们也没有讨论物理机器组。这里我们认为不能把所有机器放到一个 cell 里面。我们可以认为 TASK 就是对应容器或容器里的进程组。

## 什么是Borg？用户视角的核心概念 (2)

- **Cell:**
  - Borg中的一个计算单元的抽象，包含一组物理机器
  - 一个Cell的物理机器一定在同一个集群内 (通过Cell划分了集群)
  - 一般量级在10k个机器
- **Jobs & Tasks:**
  - 一个Job表示一个具体的计算工作，其中可以包含一系列的Task
  - Job中会对Task的运行环境进行限制，如处理器架构、OS版本、对外暴露的IP等
  - Task则对应到某个容器中的进程(组)，负责具体的计算任务

基本上 Alloc 对应了 Pod，Alloc Set 就是在 Pod 基础上做了一些扩展。

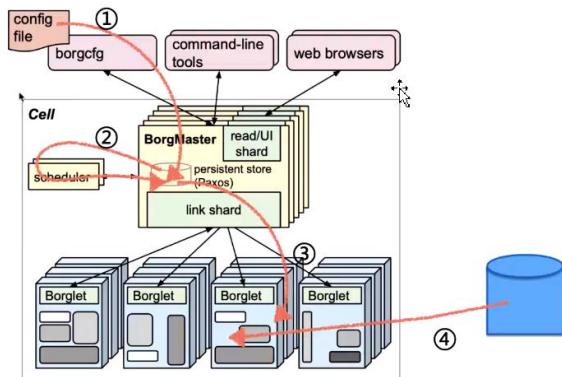
## 什么是Borg？用户视角的核心概念 (3)

- **Allocs & Alloc sets:**
  - Borg Alloc (Borg allocation的缩写) 对应是机器上预留的资源，用来运行一个或多个Task
  - Alloc在K8s继承下来，成为了Pod
  - Alloc set和Job类似，包含一组Alloc

因为我们把整个集群划分成了 cell，所以每个 cell 里会有一个 master，不同的 cell 有自己的 master。我们关注一个 cell 内部结构，这和 K8s 的结构很像。其中使用了 PAXOS 和 scheduler。

# 什么是Borg? 架构

- Cell是集群单元
- 控制面BorgMaster
  - Scheduler
  - 持久化存储
  - 命令行工具
- 数据面: Borglet
  - → kubelet



<https://vshare.sjtu.edu.cn/play/bd69c40e388ecc383872de84f36d2be7> 35:23