【计算机系统工程】绪论2：Scalability in Practice: a Highly scalable web app

@credits Yubin Xia, IPADS

在复杂系统中，web中是如何考虑可扩展性的？上节课我们讲过，从一个节点扩展到很多节点，会有很多不一样出现。整门课如果我们要有一个topic，就是scalability。

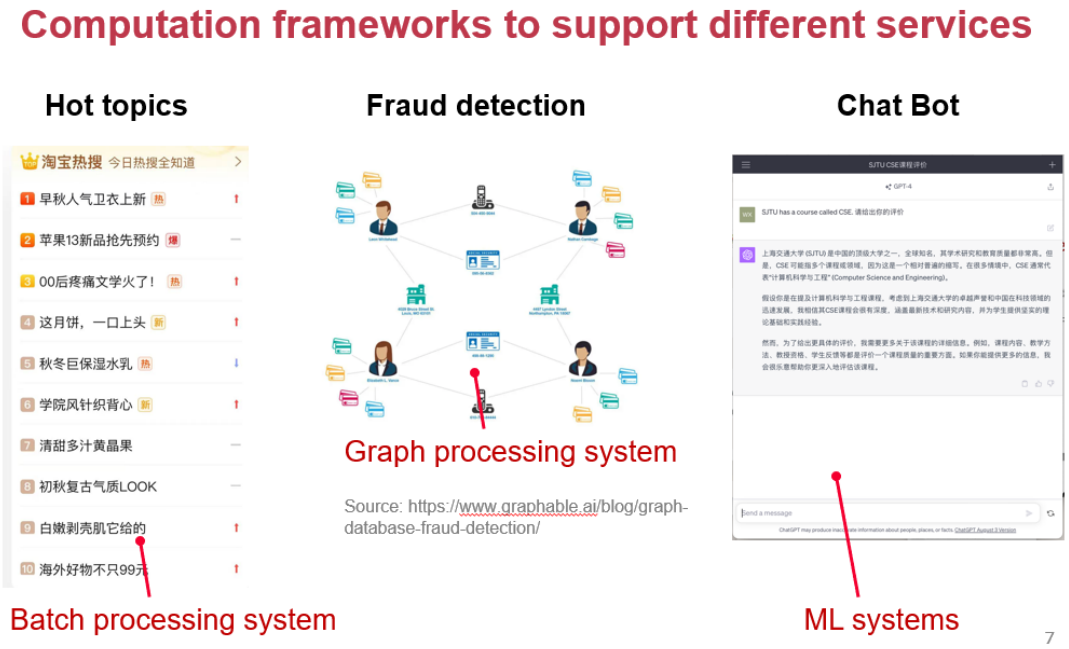
在AI时代，很多东西在底层其实和互联网时代是没有区别的。AI时代比如你问gpt一个问题，互联网时代是你下一个订单，但是底层还是一些计算资源。

> AI is just another challenging workload. The fundamental system techniques remain similar.

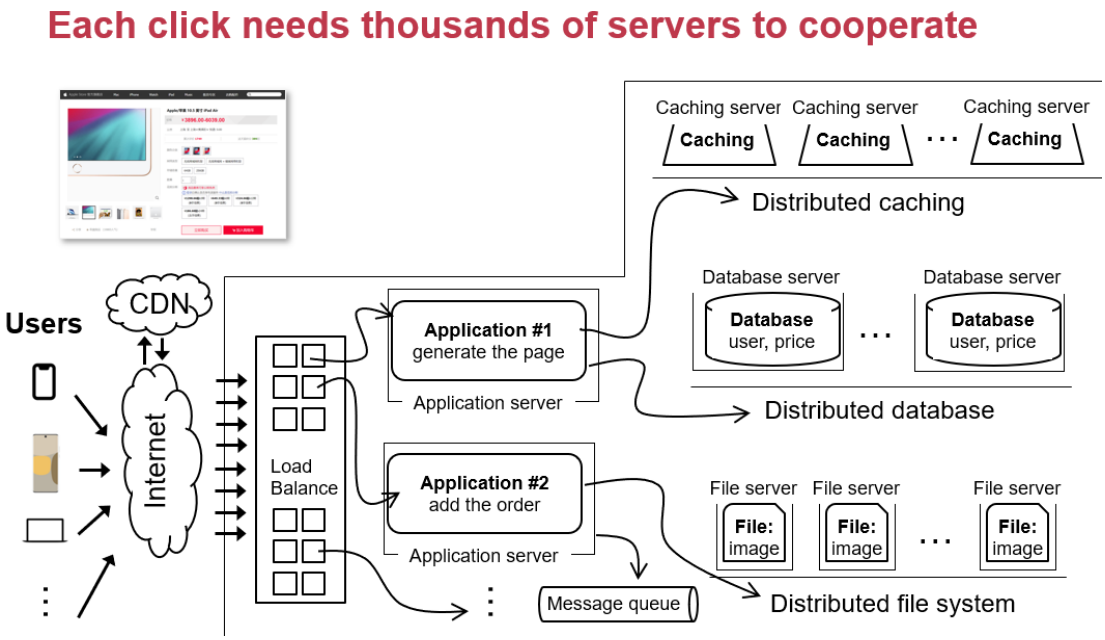
1. Case study: a e-commerce website

这个是天猫的界面：

- 产品的介绍图：图以png/jpg保存在文件系统里（因为它并没有那么结构化，因此不适合数据库）。
- 价格：经常变化，且具有统计意义，意味着它需要精确，而且需要持久化维护，且有一致性要求，因此放在database中，因为数据库对一致性有很强的保障。
- “人气”数值：人气是根据游览量等信息推算出来的值，它就不是放在database中，而是保存在key-value store里面，因为这个数值确实重要，但它的精确性并没有那么重要——你点一点它，它可能马上看上去++，但可能只是本地++，过了很久才慢慢同步回服务器上，用数据库就太浪费了。



- Batch processing 批处理：不需要实时地，比如一段时间的热搜。比如早年的map reduce。
- Fraud detection：金融交易欺诈，比如洗钱。对于这种行为，我们需要在支付的时候判断非法的概率，这显然和批处理是不同的，我要立刻处理掉可疑交易，可能只有几十毫秒的时间，而不像批处理我可以操作几个小时。这里阿里和蚂蚁用到的技术一般是图计算——data以图的形式出现，你买过什么东西、属于什么单位，做了什么交易，构成一张大图。比如我们如果在图中检测到闭环，可能就说有问题的。
- Machine-learning sys：后台会做推理，这个占据大量的时间。



每次我们点击一个按钮，后端可能有几千个服务器为我们干活。它们各司其职。

1.1 Single Machine

How to build Taobao on a single machine (in old days)?

Operating system:

- Linux (in OS class)

Serving the requests: web server

- Apache, Nginx (in ICS)

Serving the data: file system & DB

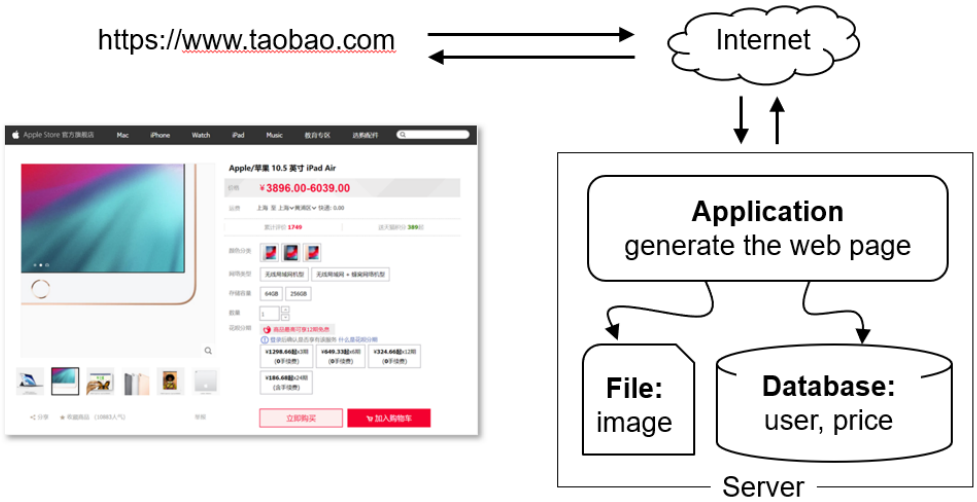
- MySQL & inode file system (in CSE)

Displaying the page: HTML

- PHP (in Web class)

LAMP = Linux + Apache + MySQL + PHP

Using one server to build Taobao



10

显然它无法很好地scale。内存最多到256GB，硬盘最多到40T，但是像Facebook，每周有十亿张图片，显然单体是做不到的，同时CPU的性能也是有极限的。

1.2 Scaling

Step #1 for scalability: disaggregating application & data

Application: handles application logic

– Can be scaled with more CPUs

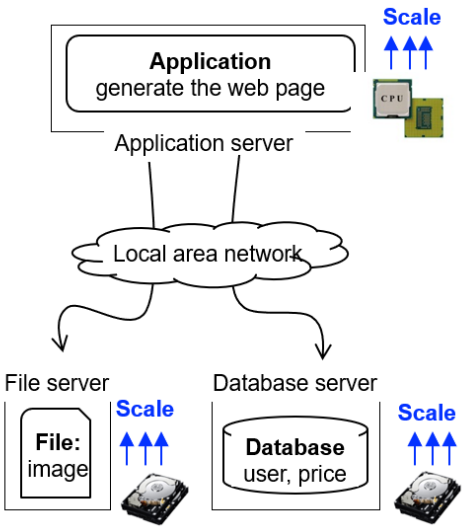
Database: requires reading/writing disk & cache

– Can be scaled with faster disks & larger memory

File system: store large bulks of data

– E.g., images, videos

– Can be scaled with faster disks



用三台取代一台，一台专门用webserver，一台fileserver，一台database。可扩展性比原来的要好。有了这个方案之后，紧接着有了一个问题，发现走网络和走本地磁盘的时延差别非常大，另外一个是database，因为database越来越大，查询变得非常慢。

Step #2 Avoid the slow data accesses? Caching

Observation: most requests access **a small portion** of the data (**locality**)

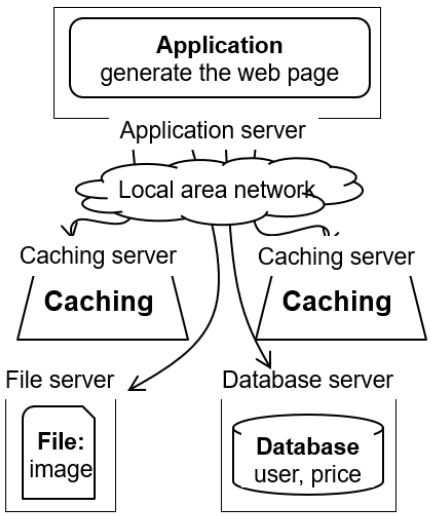
Caching system with a single node:

– E.g., page cache

– Drawbacks: **limited** DRAM capacity

Distributed caching server

– **Benefits**: huge DRAM capacity, e.g., deploy many caching servers



想出来的办法就是加一个cache——redis。但cache普遍小得多，因此我们引入多台服务器，组成分布式的缓存。

Case study of distributed cache server: Memcached

Distributed caching server

- Benefits: huge DRAM capacity

Memcached server

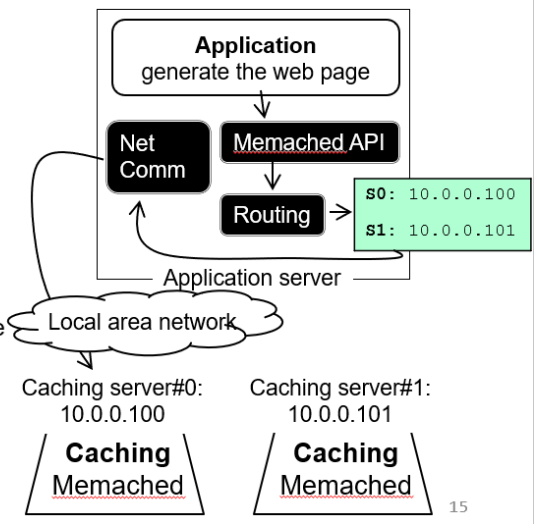
- Store all the cached data in memory
- Can scale out to use multiple servers

Memcached clients

- Check whether server has cached data
- On **cache miss**, **fallback** to database/file

Question:

- How to find which server has cached the data?



15

那我们怎么知道存在哪台服务器上？这里就要引入一台metadata server。这个会在gfs的时候再介绍。

或者我们也可以使用一致性哈希，就不需要引入一台额外的服务器：

Case study of distributed cache server: Memcached

Naïve method, hashing:

- address = key / #server

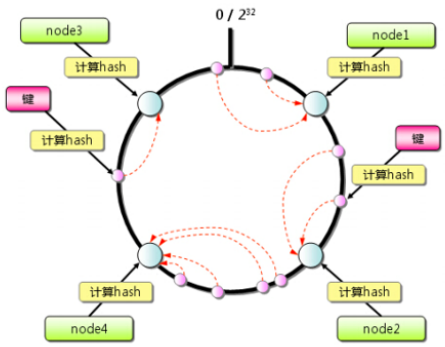
Problem:

- What if we add a new server?
- Suppose that we add server from 3→4
- Then there would be 75% miss!

Solution: consistent hashing

- Suppose that we add server from 3→4
- It will only incur 25% miss
- More details in later courses

How to find the data?



Consistent hashing

16

一致性哈希的效果就说显著降低了重新分配数据的搬移量。

Step #3 for scalability: more servers

For **stateless** application servers

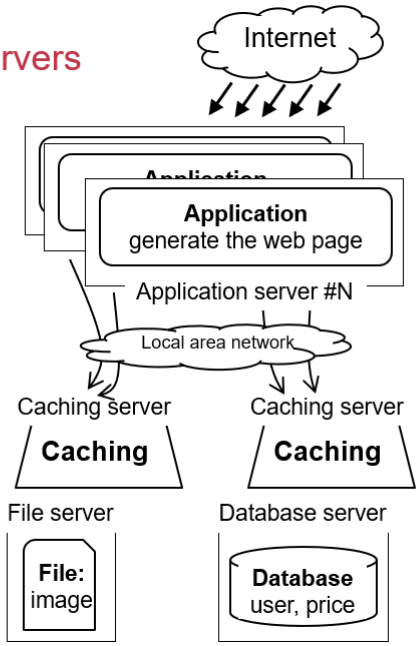
- E.g., web servers
- We can add more servers for scaling its performance

What is stateless?

- The server only executes the logic that only relies on input data but no long-term state

Benefits:

- Better fault-tolerance
- Better elasticity



这个时候存储有很多了，所以application（计算）成为了瓶颈，我们需要更多服务器来运行逻辑代码。这就出现了一个很严重的问题——不一致。

比如我有一百台服务器，我在寝室和教室连接，两次连接的服务器是不同的，所以我们的购物车可能是不一样的——这显然不行。因此，我们不能够在服务器中保存太多的状态。

于是，我们提出了所谓的stateless。任何人连到任何的服务器，我不会有任何的假设，服务器的大脑都是一片空白。这样就保证了任何服务器可以服务任何请求。这样就能很好提升容错（比如服务器炸了，用户刷新一下就完事了，对用户是无感的），也能提升灵活性。

那我的购物车放在哪里？所以我们就需要另外的服务器保存之。

Step #3 for scalability: How to do the load balance?

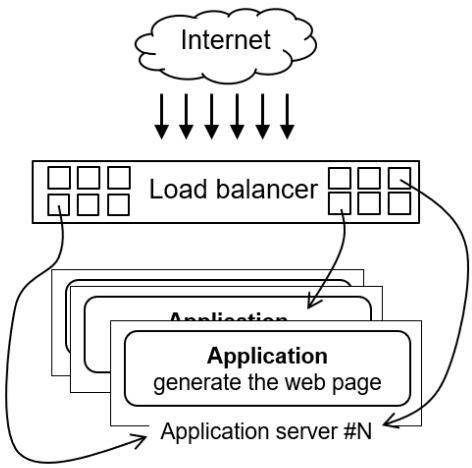
Load balance: how to route the user quests to the application servers?

Leverage the network layer

- HTTP redirection,
- reverse proxy,
- …

Load balance algorithms

- round-robin,
- random,
- hashing,
- …



18

有一万台服务器，淘宝如何选择谁来服务你？就需要负载均衡。有这样一个门户，给你在进来的时候做重定向。

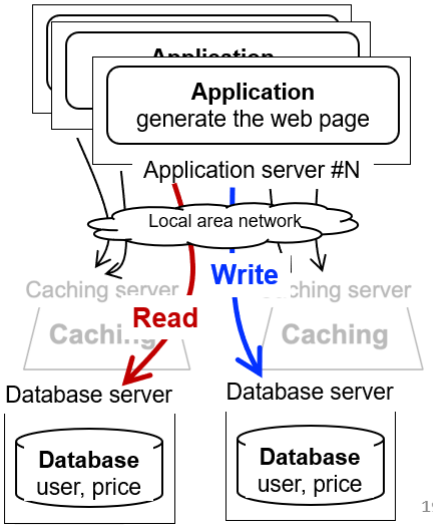
Step #4 for scalability: scaling database

#1. Separate the database for read/write

- E.g., use primary-backup replication
- The primary servers the writes and backup only serves reads

#2. Separate a table on multiple databases

- e.g., a bank has reported that a single table has **100,000,000 rows**
- However, split a table on multiple machines **complex consistency management**



19

我们现在把applicaiotn server变多了，此时database server有了瓶颈。所以我们就要去有更多的database server，如果我们简单地买两台database server存一模一样的数据，性能不会变好，但是读的时候就会更好。所以想到一个办法就是读写分离，有一个primary database只有一台用来写，secondary可以有很多台用来读。写的时候只要写一台，这个primary会逐渐把新的值发给很多台。所以主从分离。

但是还不够，因为我们发现有些时候一张table可能非常非常大，一个bank一张table有一亿行，一台机器存这个table就会有问题。有办法就是把这个table拆分成很多个table，比如十台机器保存一张table。这个拆分需要复杂的一致性管理问题。

当我们把database 的server从一台变成多台，file server也需要变成多台。

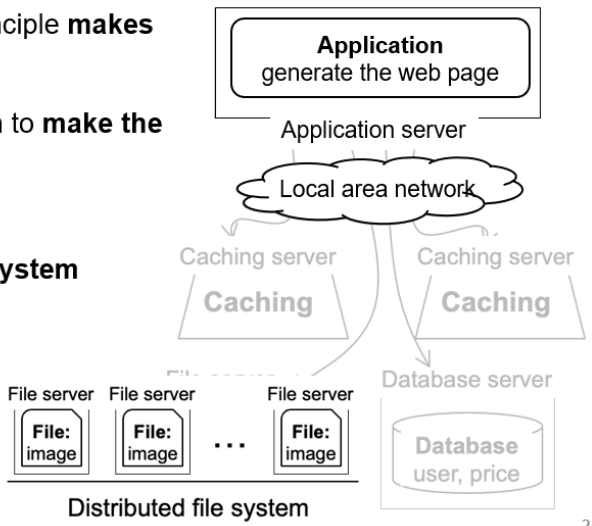
Step #5 for scalability: using distributed file system

Using multiple database in principle **makes the database distributed**

We can use a similar approach to **make the file system distributed !**

Well-known distributed file system

- NFS (Network file system)
- GFS (Google file system)
- HDFS



20

需要变成分布式的文件系统，eg:NFS,GFS,HDFS。好处文件对外的接口是统一接口，内部的文件存储形式是不可见的，1台和100台对外其他服务的暴露形式是相同的。这里就需要metadata server了。

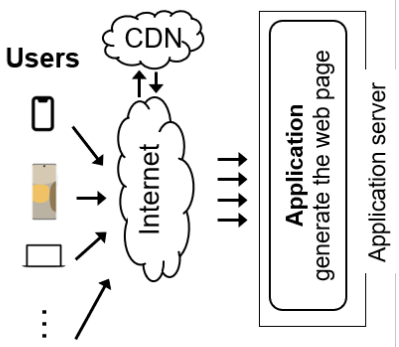
Step #6 for scalability: using CDN

Goal: return the results to user **as soon as possible**

- **Why**? Amazon has reported that **100ms** latency increase will cause **1%** financial **loss**

Core idea: caching (again)

- E.g., **CDN** (content delivery network) caches the content at the network providers, which is closer to users
- Challenge: how to do it without users' awareness?



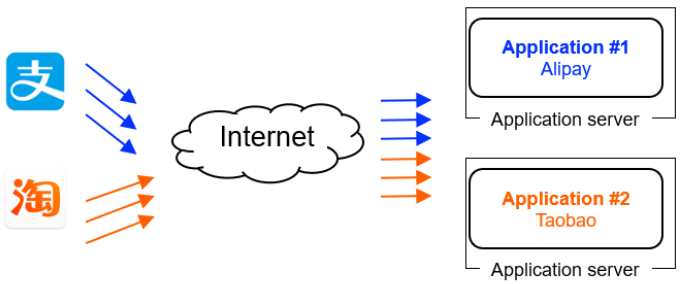
21

我们要在全国的一些地方租一些服务器，把照片这种数据缓存在这些服务器上，第二次广州的临近地区的访问就可以走临近的CDN服务器，淘宝也可以通过主动推送图片到CDN上。有了CDN cache之后，在访问database 的时候依旧可以从淘宝的服务器上下载，在访问视频的时候就可以从cdn上去访问。在淘宝的webserver中，图片其实是CDN的地址，发到我们计算机上以后，再会从获取的CDN路径上下载图片。我们可以进一步地减少时延。

Step #7 for scalability: separate different applications

Use dedicated servers for different applications

- E.g., micro-services



22

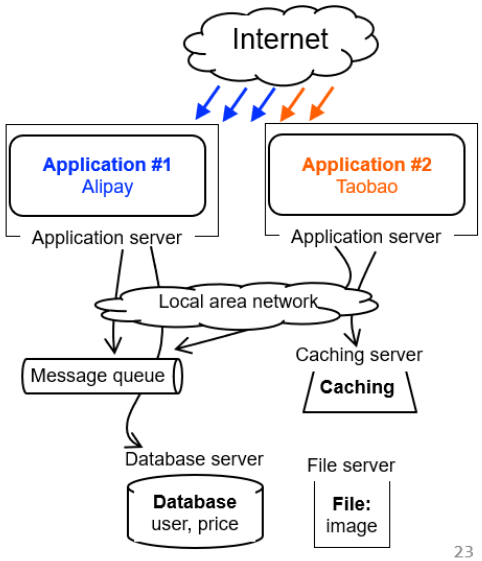
支付宝和淘宝用一样的webserver稳定性相对更差，专用服务器性能会高。所以淘宝和支付宝打散，各自用服务器。但我们发现其实有些是可以复用的，很多可以merge在一起。这就是阿里之前一直在提倡的数据中台（但现在似乎已经挂了，说小企业别搞这些，但现在又出来一个fashion word，所谓的平台工程，platform engineering，anyway它不重要）。

How different applications communicate? MQ or DB

Message queue (MQ): applications send the message the queue, and the queue can buffer the message (somewhere between RPC and database)

– E.g., Apache Kafka

Or, applications can directly use the **databases, caching** (e.g. KV) **or file system to communicate with shared data**



23

微服务，服务和服务之间的通信变得非常关键——比如，MQ——kafka！

How to handle complex requests?

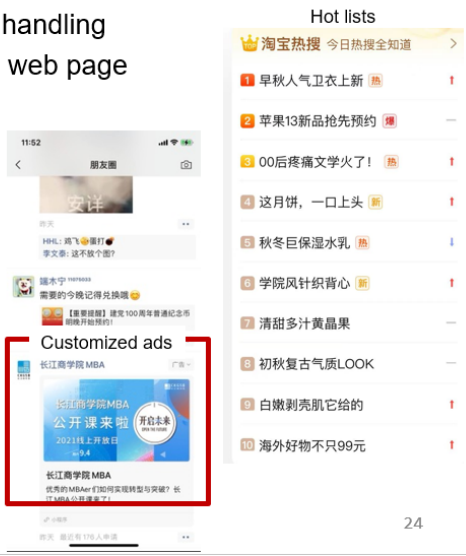
Example: after the website becoming larger, handling requests is far more than displaying a (static) web page

Alipay (支付宝)

– E.g., fraud detection

Taobao

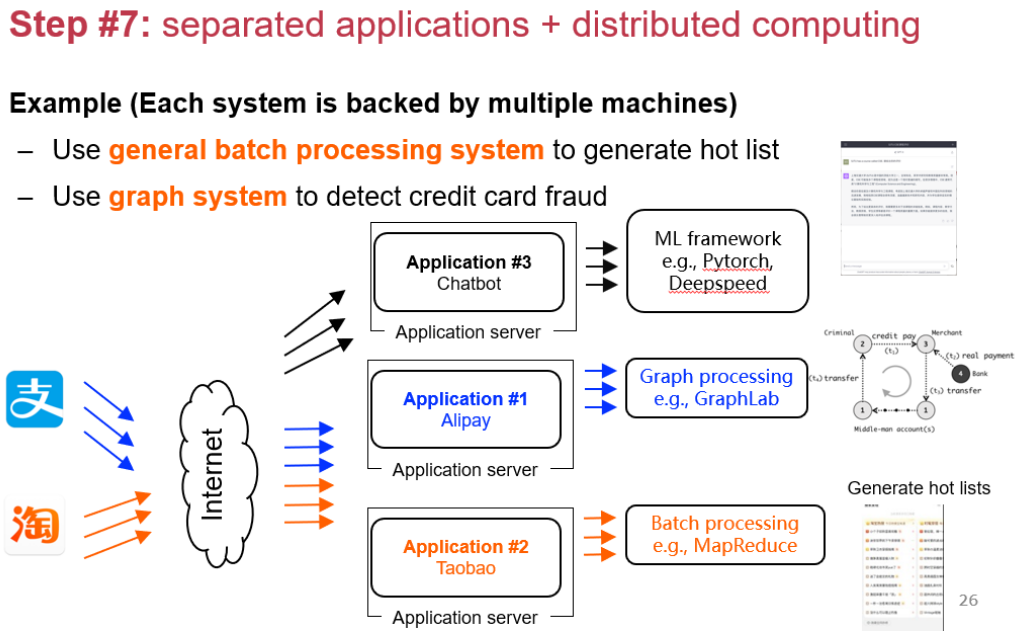
– Hot list (热榜)
– Dynamic product ads (千人千面)
– Recommendation (you might also like)
– …



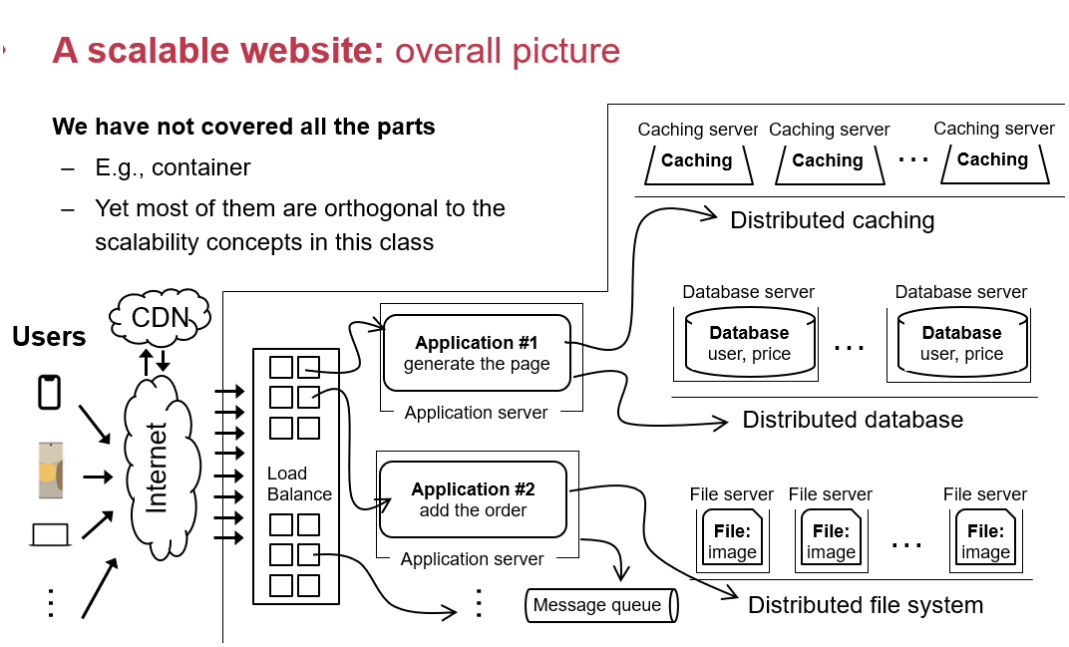
24

我们如何处理复杂请求？比如支付宝的金融欺诈，还有淘宝的千人千面，都是比较复杂的（千人千面就是每个人每次打开的页面都是不同的）。当然千人千面这种服务意味着大量的运算。

所以我们就需要一些分布式计算的framework，比如mr做批处理；对于信用卡欺诈，需要用到图计算graphlab、pregel。这些就是后端长时间运行的服务。总之也是以服务器的形式在跑，机器学习也是插了A100的服务器嘛。



最后组合在一起，就形成了最终本门课的地图：



分布式系统：

> a collection of independent/autonomous connected through a communication network working together hosts to perform a service

- 互相自治、独立
- 通过网络连接：一旦用网络，就要花很大的精力来讨论容错。

背后是几十万台服务器！

Datacenters that power the scalable website

Large-scale distributed systems: 10K – 100K servers

- Each rack: 40 servers
- Network: 10Gbps – 100Gbps in rack
- 10-100 MW of power

Source: "The Datacenter as a Computer --- An Introduction to the Design of Warehouse-Scale Machines"

32

1.3 Fault

Fault is common: fault, error, failure

Fault can be latent or active

- if active, get wrong data or control signals

Error is the results of active fault

- e.g. violation of assertion or invariant of spec
- discovery of errors is ad hoc (formal specification?)

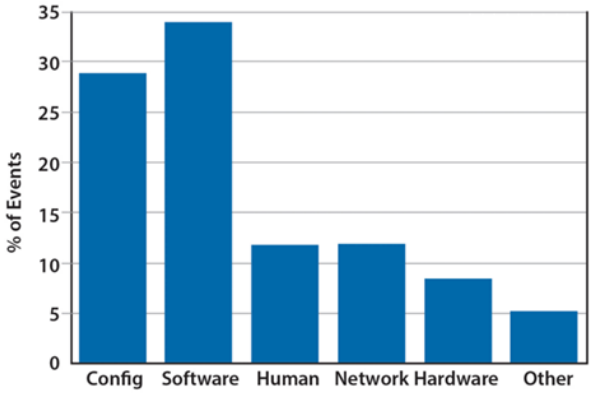
Failure happens if an error is not detected and masked

- not producing the intended result at an interface

Fault是错误的原因。Error是fault的结果，failure就是很大的失败。当我们的系统不出错的情况，一百万台机器都不出错是不太可能的。因为总体中有一台机器的出错率是随着机器数量指数增加的。

- Fault就是单车车胎扎了一个钉子；
- Error就是车胎漏气了（当然如果没漏气，钉子即使扎在那里，也没造成error）；
- Failure就是因为扎了个钉子，你去赶着考试，结果迟到了16分钟。

Fault：人、软件bug、硬件问题、断电、自然灾害。比如宇宙射线翻转bit，但这个bit没人用，就没事。



Fault is common especially in **large distributed systems**

What are the causes?

Why faults are common especially in distributed systems? Scale!

– "Suppose a cluster has ultra-reliable server nodes with a stellar mean time between failures (MTBF) of 30 years (10,000 days)—a cluster of 10,000 servers will see an average of one server failure per day. "

Causes:

– Operation error (human, configuration, etc.)
– Software error (e.g., bug)
– Hardware error
– Power outage
– Natural disaster

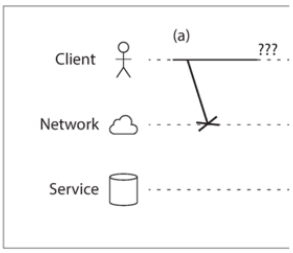
> 再举个例子，希捷的硬盘，平均无故障运行时间，官网上的数据假设说是30万小时，但可能这个公司本身都没有30万小时。所以如何测出的？拿1000个硬盘，同时运行3000小时，出了10个错误，就得到30万小时。

我们不可能打造一个所有机器不出错的分布式系统，我们希望用不可靠的组件组合在一起组成一个可靠的分布式系统。我们希望同时断掉90%机器的电，其他的机器还能work，**整个系统还以一种奇怪的方式运行（比如性能降低十倍）**（这就是所谓的**灰度错误**，它不是黑或者白——你一头雾水，怎么突然变慢了，慢的原因有很多啊）。anyway，这就是容错能力，但是又有一堆问题。

Faults and partial failures

Example: unreliable network

- A client (e.g., Smartphone), sends requests to the service (e.g., Taobao), through network (5G, Wifi)
- The client gets: “网络竟然奔溃了”, how can a network break?



40

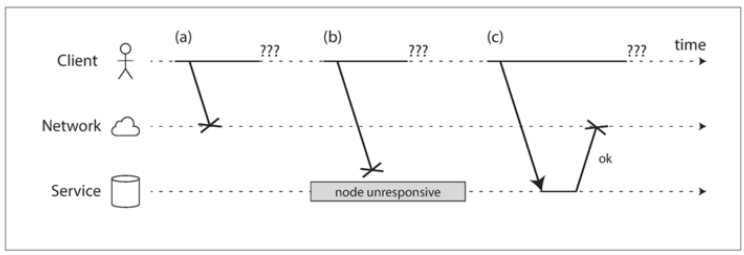
网络是非常令人头疼的：当client发包但是没有收到回复，你永远不知道别人是收到了不回还是没收到！

- 如果是没收到，那这件事等价于没发生过；
- 如果收到了但是ack掉了，那就很麻烦。比如你转账一百元，你又转了一次，恭喜你转账两百元成功！

Example: unreliable network

A user sends a request but **the server does not reply**, possible reasons:

1. The request may have been **lost** (e.g., someone unplugged a network cable).
2. The request may be waiting in a queue and will be delivered **later** (e.g., the network or the recipient is overloaded).
3. The remote node may have **failed** (e.g., it crashed or it was powered down).



41

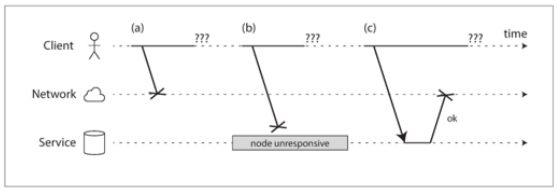
Example: **unreliable network**

A user sends a request but **the server does not reply**, possible reasons:

4. The remote node may **have temporarily stopped** responding (e.g., it is experiencing a long **garbage collection pause**)

5. The remote node may have processed your request, but the **response** has been **lost** on the network (e.g., a network switch has been misconfigured).

6. The remote node may have processed your request, but the response has been **delayed** and will be delivered later (e.g., the network or your own machine is overloaded).



42

Another common fault: network partition

A **network partition** refers to network decomposition into relatively independent subnets

- Can happen when a switch is being upgraded in a datacenter
- Can even happen when being attacked by sharks

Network partition is usually a reality

- You can never count on the network connectivity

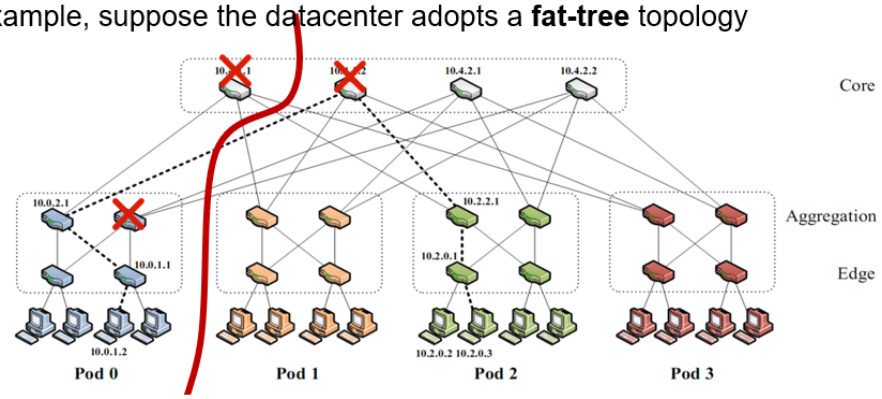
这个故障就是所谓的split brain。

比如杭州和北京之间通路很通畅，挖掘机一铲子把杭州和北京的光纤挖断了。结果杭州的人访问杭州的，北京的人访问北京的，都没问题。问题就造成了杭州的人看不到北京同学的朋友圈。

我支付宝在杭州花了一百块钱，赶紧润到北京，这一百块钱还能再花一次！

为什么有这种事情？因为其实很多时候我们数据中心是这种布局：

For example, suppose the datacenter adopts a **fat-tree** topology



45

这三台一旦挂掉，整个网络就分裂了，两部分失去通信了。关于网络里面可能出现更阴间的问题，比如光缆被鲨鱼咬断了：

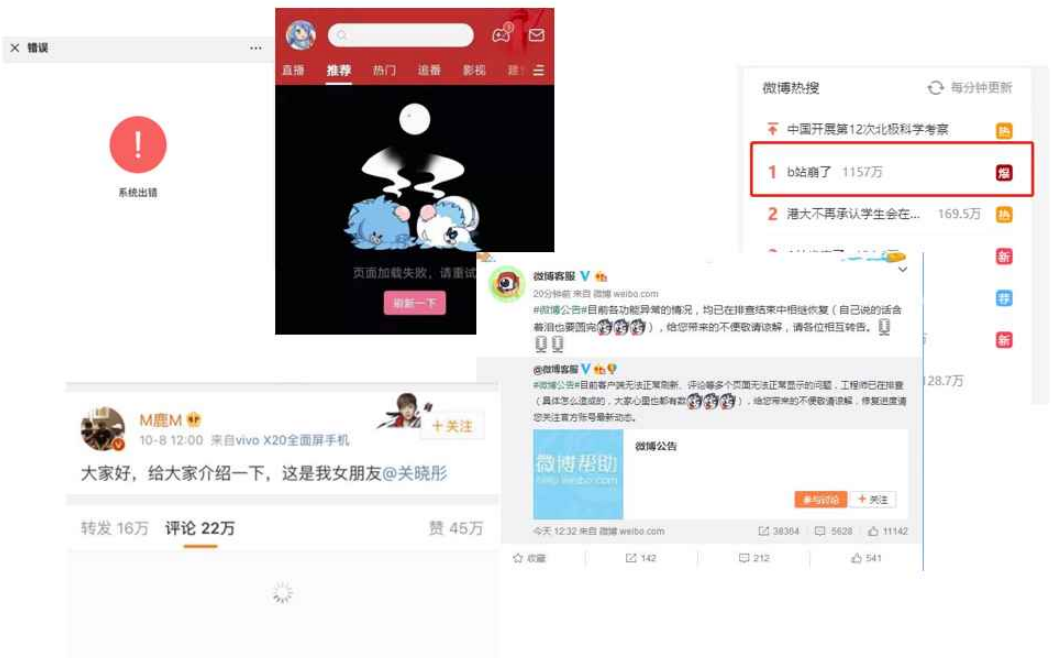
Another common fault: network partition

A **network partition** refers to network decomposition into relatively independent subnets

- Can happen when a switch is being upgraded in a datacenter
- Can even happen when being **attacked by sharks**

分布式系统里面就是不知道出了什么阴间的错误，说不定就崩了。

> You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done.
>
> - *Leslie Lamport*



1.4 Availability and reliability

对于可靠性的评价方法，可以用可靠性，有几个9，也就是99.9%，每年允许8小时网站不work。

Availability and reliability

Availability: A measure of the time that a system was usable, as a fraction of the time that it was intended to be usable (x nines), corresponding **downtime**:

- e.g. 3-nines -> 8 hour/year
- e.g. 5-nines -> 5 min/year
- e.g. 7-nines -> 3 sec/year

Metrics to measure reliability

- MTTF: mean time to failure
- MTTR: mean time to repair
- MTBF: mean time between failure
- **MTBF = MTTF + MTTR**

$$MTTF = \frac{1}{N}\sum_{i=1}^{N} TTF_i$$

$$MTTR = \frac{1}{N}\sum_{i=1}^{N} TTR_i$$

MTTF就是mean time failure，连续多久不宕机。MTTF时间非常长不意味着availability就会好。

- 比如debug了一个bug一个礼拜；实际availability只有50%。
- 同样，每分钟出一次错，每次修复一毫秒，这个显然是更好的。

- MTTF：连续多久不宕机的平均时间。
- MTTR：出错以后修复的平均时间。
- MTBF：两次出错的平均间隔时间。

MTBF = MTTF + MTTR

事实证明，我们可以得到高的可用性。微信、百度这种我们很少遇到出问题的情况，说明它的MTTF是相当长的。在技术上怎么实现高的可用性呢？

就是rudueancy，也就是冗余，单台机器挂了之后其他机器可以顶上，我们不是为了构造一台永远不会出错的机器，而是一旦挂了就替换出去。

> 所以大家也可以经常间隔一段时间换一台笔记本，然后再切换回去，有助于保证笔记本的stateless。

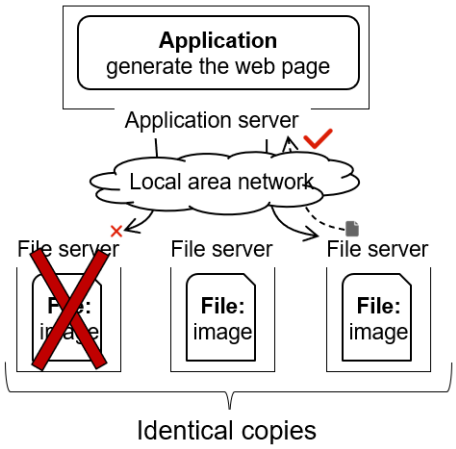
Achieving high availability: handling failures w/ replications

Replication

– replicas: identical multiple copies

Example: **replicated file servers**

– If one copy survives, the application is available



去阿里云买存储的时候我们就会发现，一备份和三备份价格是有差距的。

这个时候新的挑战就是consistency!

Challenge: consistency

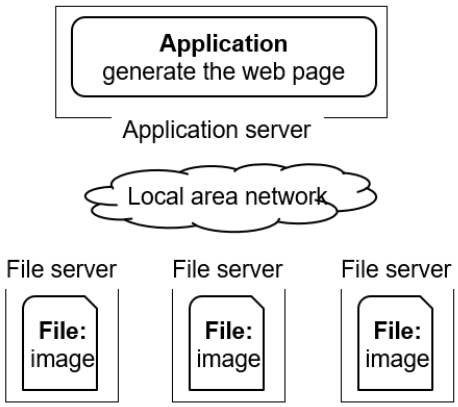
Replication

– replicas: identical multiple copies

Example: **replicated file servers**

– If one copy survives, the application is available

Challenge: consistency



Achieving high availability: handling failures via retry

Restart or **reconstruct**

– Monitoring and catching errors

– Restart or reconstruct the system (sub-system)

– E.g., restart the stateless application server is ok

What about consistency? Must made trade-off

– Stateless applications does not have consistency issue

– Some applications, like **Google search**, can even tolerate occasional inconsistency

- Can you notice the inconsistency of search results?

1.5 The CAP theorem

The CAP theorem: 2 out of 3

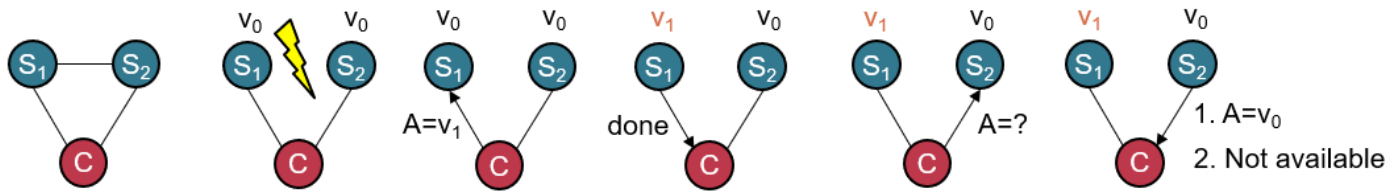
It is impossible for a distributed computer system to simultaneously provide all three of the following guarantees

- **Consistency** (all nodes see the same data at the same time)
- **Availability** (a guarantee that every request receives a response about whether it succeeded or failed)
- **Partition tolerance** (the system continues to operate despite arbitrary message loss or failure of part of the system)

CAP定理：一致性（Consistency）、可用性（Availability）、划分的容错（Partition）三者只能选二。

Partition tolerance：就算网络被鲨鱼咬了，一个网络变两个网络了，也能正常运行。CAP这三个同时只能满足两个。

> 我们可以用下面这个例子来简单的证明一下：
>
> 亚马逊的业务有两个区域：美国区和欧洲区。美国的用户连着美国区，而欧洲用户连接欧洲区。
>
> 一致性：所有用户看到的商品剩余数量是真实的、一'd。
>
> 可达性：用户看到商品库存为1，实际上已经没有了，就说明availability有问题。



> 假设有两个server一个client。假设partition了，c去s1写了A=v1，然后去s2读的时候：
>
> - 发现读出来是v0，consistency挂了。（但是保证了availability）
> - 服务器告诉你我不知道，那就是availability挂了，consistency实现了。
>
> 在CAP中，我们能牺牲P吗？这个很难，因为P这个东西不是你可以决定的。一铲子把光纤挖断掉是不可预估的，所以P往往是一个前提，所以我只剩下AP和CP。

Partition Tolerance

"P" is usually a reality

– You can never count on the network connectivity

AP: sacrifice C

– If you have one book but sell it to two customers

– Maybe just an apology and a small gift coupon

CP: sacrifice A

– If you are selling train ticket but cannot deliver

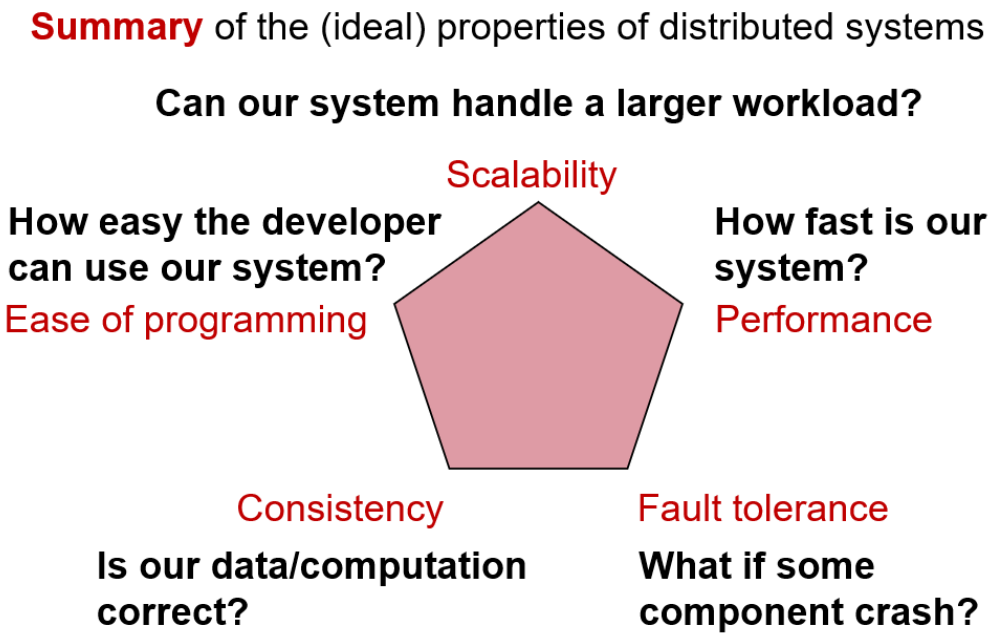
– Customer may sue you

– But the user can fail to buy the ticket, e.g., 12306 during spring festival in the last few years

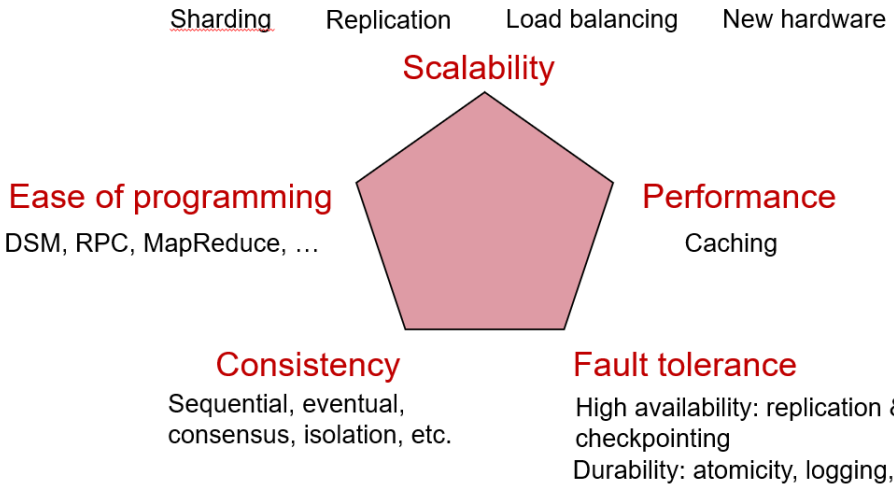
如果我们选AP放弃了C，对于淘宝这种场景是可以接受的。淘宝牺牲了C，最坏情况下是一本书卖给了两个人。支付宝选的是CP，最后付了多少钱一定是一样的。而淘宝选的是AP，因为要优先保证可用性，不一致的情况下相对可以接受，最多就是货量不足不发货了。

在Amazon上有一个功能是一键购买，也就是没有付款流程，点一下就能买好了。也就是**亚马逊利用用户那一瞬间的冲动，所以亚马逊是不可能放弃A的**。所以我们要考虑业务场景下，来选择AP和CP。在12306一张票卖给两个人，这是不能接受的，所以需要CP。

Take away message



Summary of the (ideal) properties of distributed systems



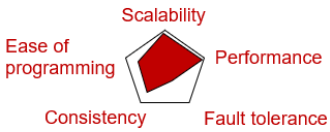
不存在完美的系统：

Summary of the (ideal) properties of distributed systems

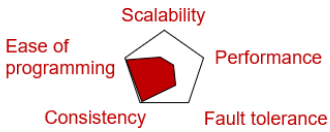
These properties typically cannot achieve at the same time

- The adults want them all, but the reality forces them to make trade-offs

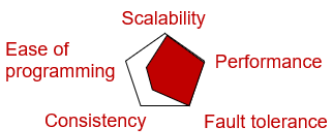
Remote Procedure Call (RPC)



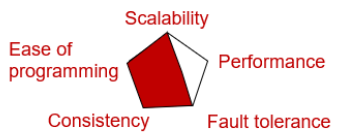
Distributed Shared Memory (DSM)



NoSQL databases



NewSQL databases (e.g., Spanner)



这门课才有了存在的意义，我们知道某个场景选用合适于之的系统。