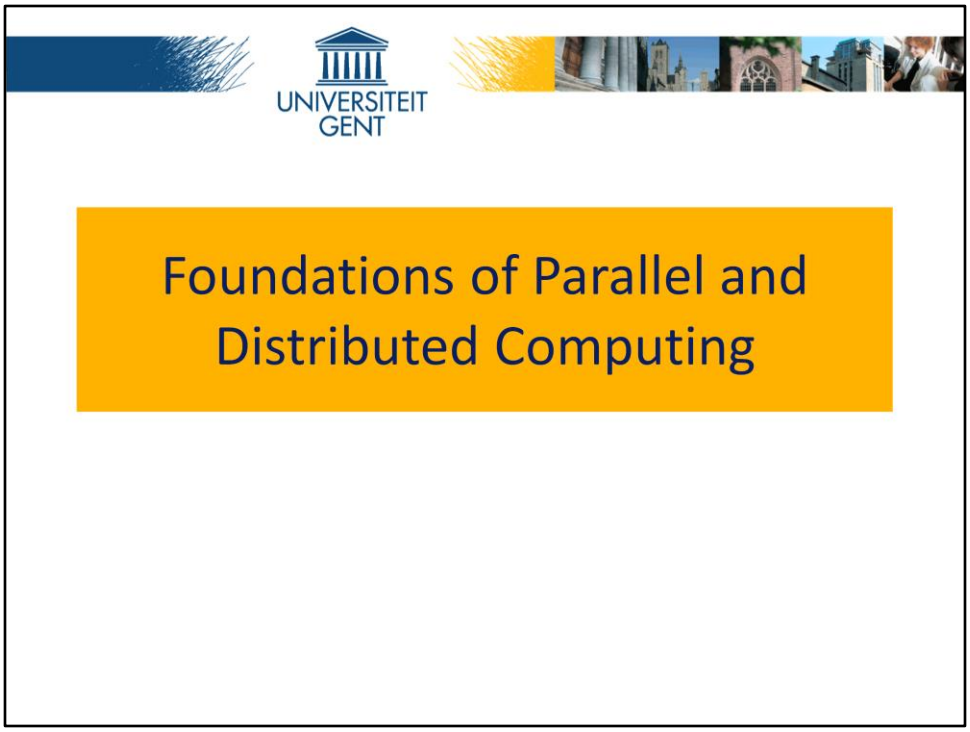
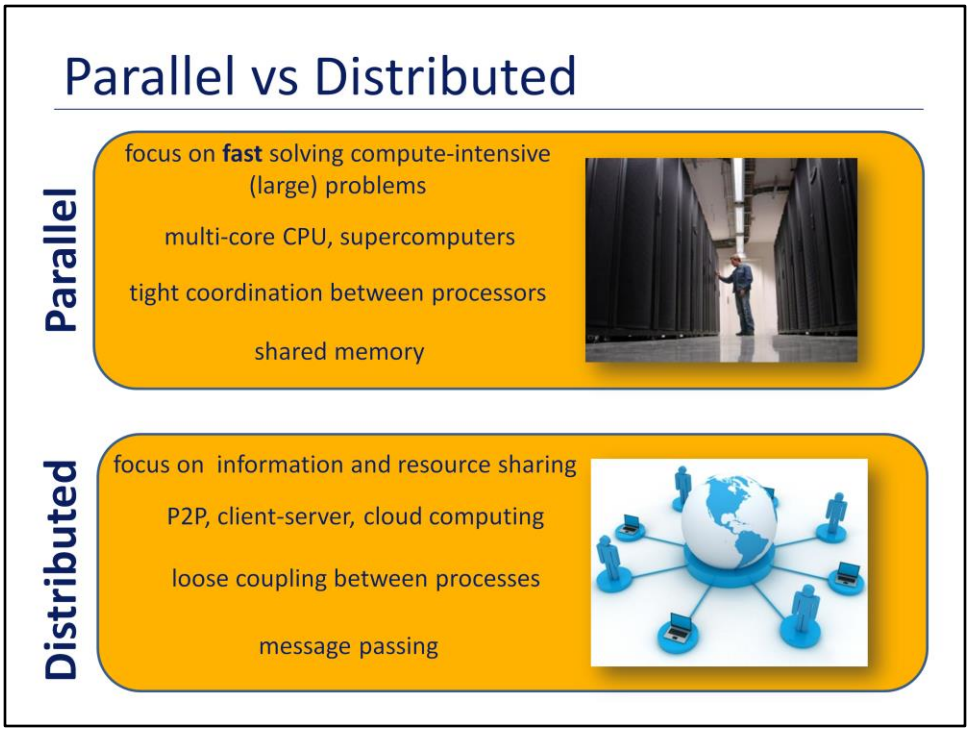

UNIVERSITEIT
GENT
Foundations of Parallel and
Distributed Computing



The difference between parallel and distributed computing is often vague. Many systems leverage on principles from both paradigms simultaneously. In broad terms, the goal of parallel processing is to employ all processors to perform one large task. In contrast, each processor in a distributed system generally has its own semi-independent agenda, but for various reasons, including sharing of resources, availability, and fault tolerance, processors need to coordinate their actions.

Parallelism is generally concerned with accomplishing a particular computation as fast as possible, exploiting multiple processors. The scale of the processors may range from multiple arithmetical units inside a single processor, to multiple processors sharing memory, to distributing the computation on many computers. On the side of models of computation, parallelism is generally about using multiple simultaneous threads of computation internally, in order to compute a final result.

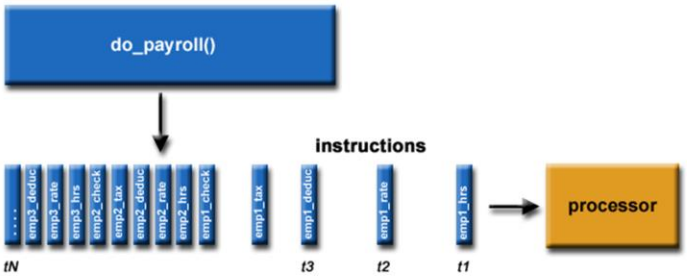
Distributed computing studies separate processors connected by communication links. Whereas parallel processing models often (but not always) assume shared memory, distributed systems rely fundamentally on message passing. Individual entities are often loosely coupled. The model is automatically used when independent entities cooperate (e.g. a web shop provider contacts a payment provider when a customer is finalizing his order). It is also used in cloud computing, where different application components reside in separate virtual environments. Failure (of processor nodes or communication links) is a normal situation.

Outline

- Parallel computing
  - serial vs parallel
  - the need for speed: why parallelism?
  - theoretical boundaries to maximum speed-up
- Distributed computing
  - what is distributed computing
  - CAP theorem and its extensions
  - RAFT consensus algorithm

Serial computing

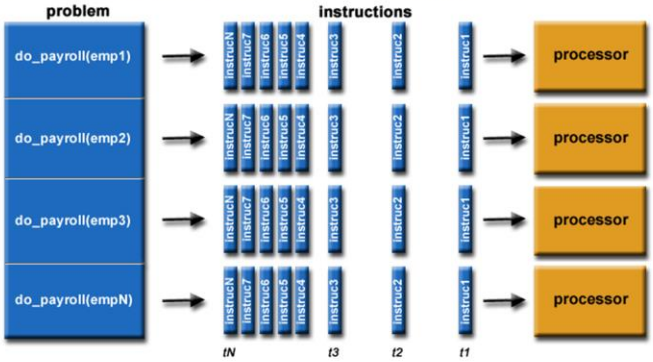
- Discrete series of instructions executed sequentially
- Executed on a single processor
- Only one instruction executed at any time



Parallel computing

Simultaneous use of multiple compute resources to solve a computational problem

- Problem decomposed in parts that can be solved concurrently
- Instructions from each part execute simultaneously and are combined afterwards
  - Note: concurrent computing: IPC *during* task execution
- An overall control/coordination mechanism is employed



Why parallelism?

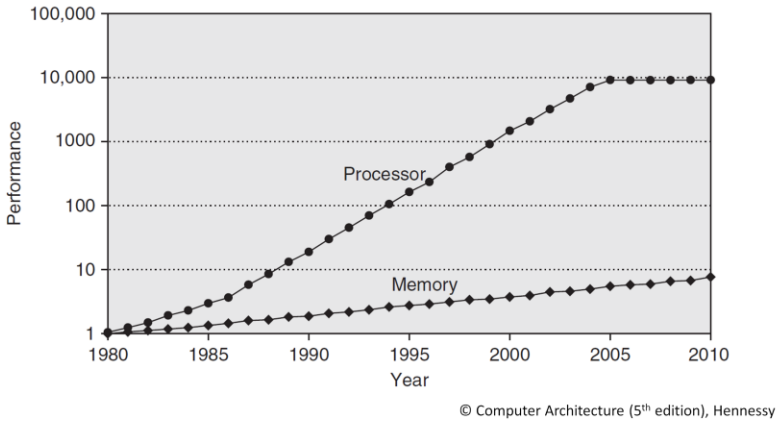
- … it incurs additional troubles:
  - need to rewrite programs
  - synchronization and control
- Three walls to serial performance (=single CPU speed)
  - Memory wall
  - ILP wall
  - Power wall
- Not walls, but increasingly steep hills to climb

For general-purpose processors, much of the motivation for multi-core processors comes from greatly diminished gains in processor performance from increasing the operating frequency. This is due to three primary factors:

- The *memory wall*; the increasing gap between processor and memory speeds. This, in effect, pushes for cache sizes to be larger in order to mask the latency of memory. This helps only to the extent that memory bandwidth is not the bottleneck in performance.
- The *ILP wall*; the increasing difficulty of finding enough parallelism in a single instruction stream to keep a high-performance single-core processor busy.
- The *power wall*; the trend of consuming exponentially increasing power with each factorial increase of operating frequency. This increase can be mitigated by "shrinking" the processor by using smaller traces for the same logic. The *power wall* poses manufacturing, system design and deployment problems that have not been justified in the face of the diminished gains in performance due to the *memory wall* and *ILP wall*.

Memory speed

Growing disparity between CPU and memory outside the CPU chip

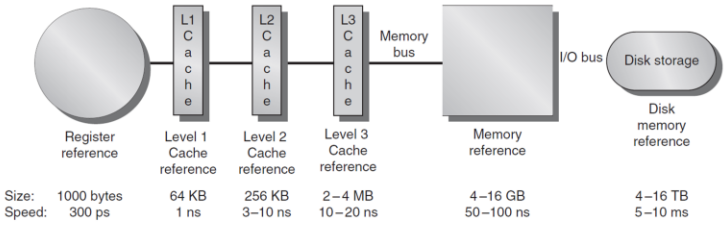


The advance in performance of processors has increased the importance of the memory wall. The graph plots single processor performance projections against the historical performance improvement in time to access main memory. The processor line shows the increase in memory requests per second on average (i.e., the inverse of the latency between memory references), while the memory line shows the increase in DRAM accesses per second (i.e., the inverse of the DRAM access latency).

Starting with 1980 performance as baseline, the gap in performance as the difference in time between processor memory requests (for a single processor or core) and the latency of a DRAM access is plotted over time. Note that the vertical axis is on a logarithmic scale.

Memory wall

- A system (CPU-memory duo) can't move at CPU top speed
- Improvements?
  - memory bandwidth: more pins (32 → 64 bit →…)
  - memory latency: even more challenging!
    - caches are fast, but limited in size



Even when the clock frequency of CPU cores keeps improving, we cannot fully utilize this because of the memory wall. The memory wall is caused by two factors:

- **Memory bandwidth**: how much data can be transferred per second between CPU and memory. Improving the bandwidth requires to add more "pipes", e.g. more pins that come out of the chip for the DRAM for example. This is challenging in terms of technology since the real estate on a chip is limited.
- **Memory latency**: the amount of time it takes for an operation to complete. For example: the time needed for a CPU to complete a 32 bit write/read operation (to off-chip memory). This is even harder to improve than bandwidth. While improving bandwidth can (in principle) be realized by doing more of the same; improving latency requires breakthroughs in material sciences (e.g. optical communication).

Instruction Level Parallelism wall

- Basic idea
  - Overlap execution of independent instructions
  - Work on many instructions during the same clock cycle

```
for(int i=0; i < 1000; i++)
  x[i] = x[i] + y[i];
```

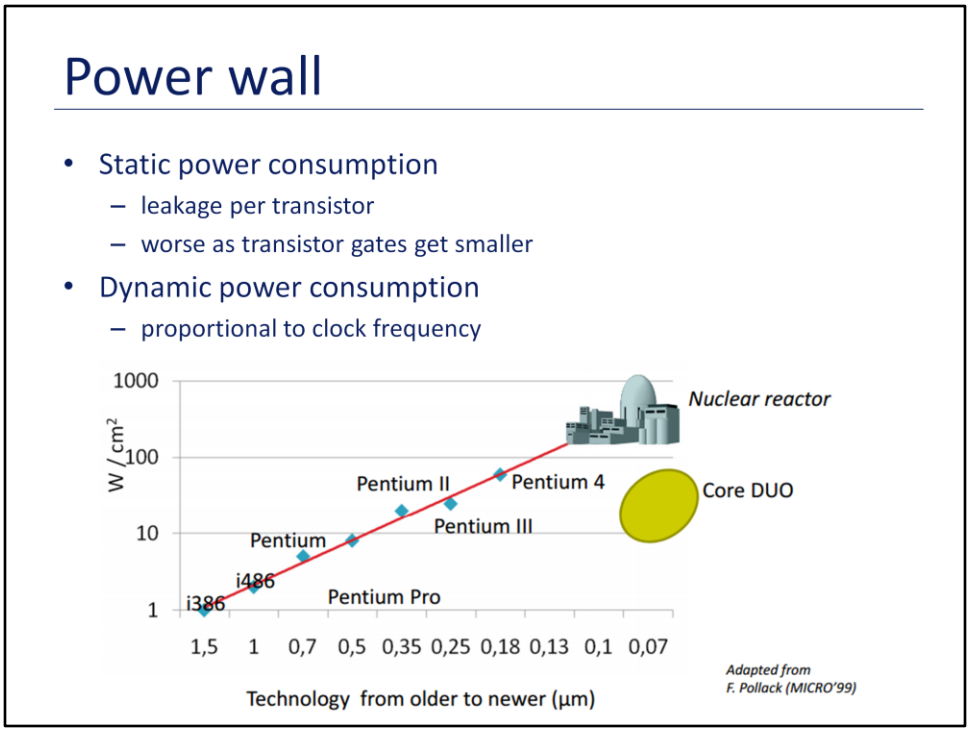
```
e = a + b;
f = c + d;
g = e*f;
```

- How?
  - Instruction pipelining, out-of-order execution, speculative execution, superscalar execution
- But… acceleration using ILP is plateauing
  - You need large blocks of instructions that can run in parallel
  - "speculation" success difficult to predict
  - super-linear increase in complexity without linear speedup

Instruction Level Parallelism aims to increase the throughput of the number of instructions executed per unit of time (clock cycle) by a single core. Various techniques are used:

- Instruction pipelining: divide each instruction into series of sub-steps (micro-operations) so that the execution of multiple instructions can be partially overlapped
- Out-of-order execution: instructions execute in any order but without violating data dependencies
- Speculative execution: allows the execution of complete instructions or parts of instructions before being sure whether this execution is required
- Superscalar execution: multiple execution units are used to execute multiple instructions in parallel

However, the upper limits of the acceleration through ILP exploitation are almost reached. First, for ILP to be efficient, you need large blocks of instructions that can be [attempted to be] run in parallel. This is not always the case and you cannot go beyond your critical path (see later). Second, ILP is often based on speculation: if you guessed wrong, you throw away that part of your result. Third, data dependencies may prevent successive instructions from executing in parallel, even if there are no branches.

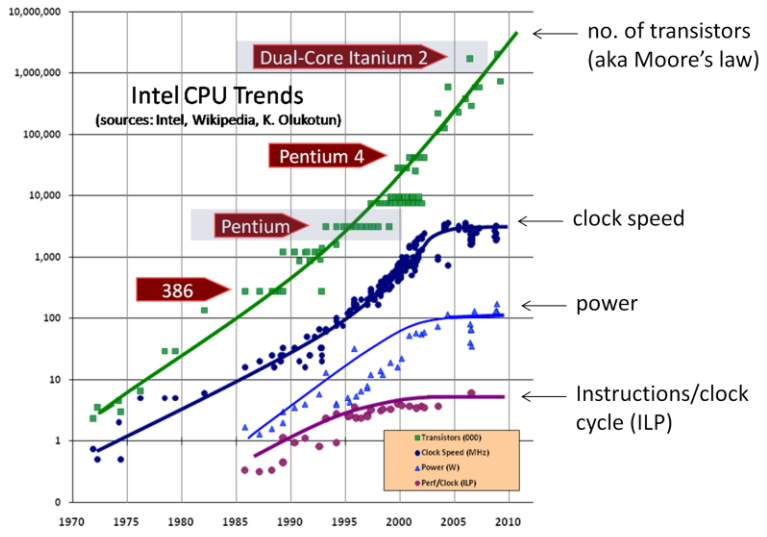


Power, and not manufacturing, limits improvements in the speed of individual cores. The power consumption of a core is the sum of two contributions:

- **Static power consumption:** this is mostly leakage, it is the power dissipated by a transistor whose gate is intended to be off. Leakage power dissipation gets worse as transistor gates get smaller, because gate dielectric thickness must proportionally decrease.
- **Dynamic power consumption:** this is the part that is related to the amount of work executed by a single core. The dynamic power consumption is proportional to (a.o.) the clock frequency.

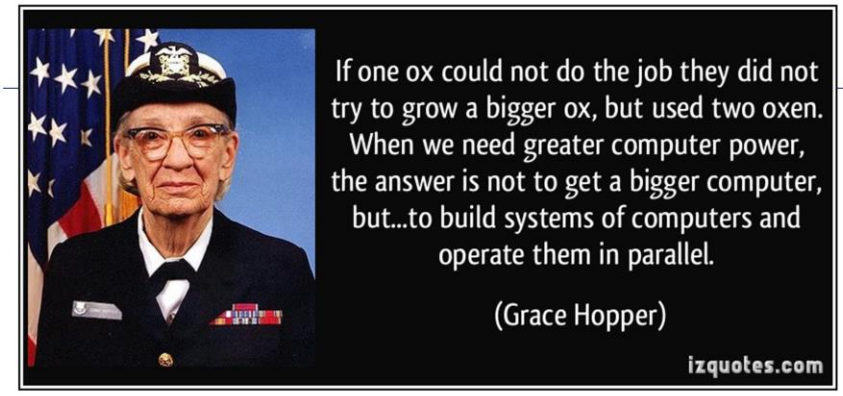
For this reason, the increase in INTEL CPU clock speed was already stopped in 2007. From then on, they switched to multiple cores.

Sequential execution has lost steam



The three walls discussed in the previous slides indicate that the model of sequential execution has lost momentum. As chip geometries shrink and clock frequencies rise, the transistor leakage current and the dynamic power increases, leading to excess power consumption and heat. Also, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies.

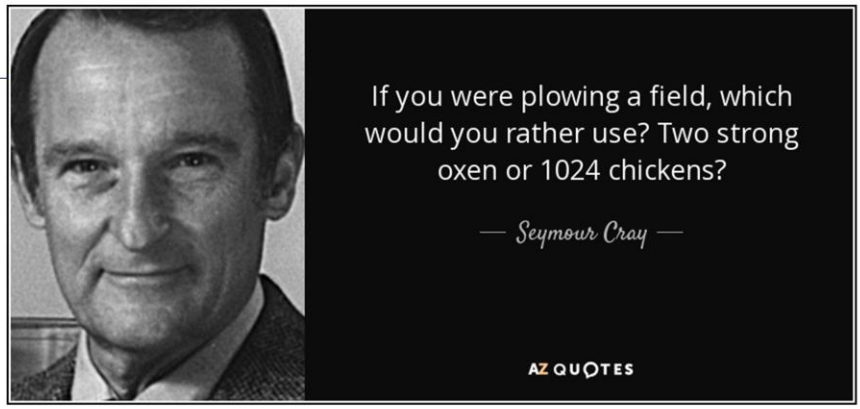
On the above graph, there is however one bright spot: the number of transistors per unit area is still increasing. This is the well-known Moore's law.



Sacrificing uniprocessor performance for power savings can save you a lot

Example:

- Scenario One: one-core processor with power budget W
  - Increase frequency/ILP by 20%
    - Substantially increases power, by more than 50%
    - But, only increase performance by 13%
- Scenario Two: Decrease frequency by 20% with a simpler core
  - Decreases power by 50%
  - Can now add another core (one more ox!)

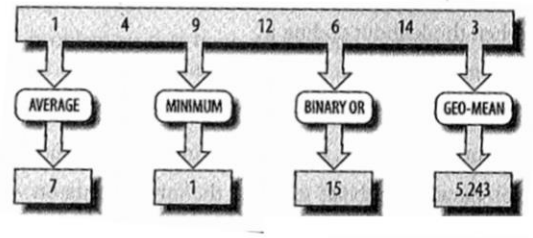


For certain classes of applications (not including field plowing…) you can run many cores at lower frequency and come ahead (big time) at the speed game.

Types of parallelism

- Task parallelism
  - entirely different calculations on either the same or different sets of data
  - allocate subtasks to a processor
- Data parallelism
  - same calculation is performed on the same or different sets of data
  - allocate subset of data to process
- Pipelining: hybrid data/task parallelism
  - a parallel pipeline of tasks, each of which might be data parallel
  - e.g. image processing

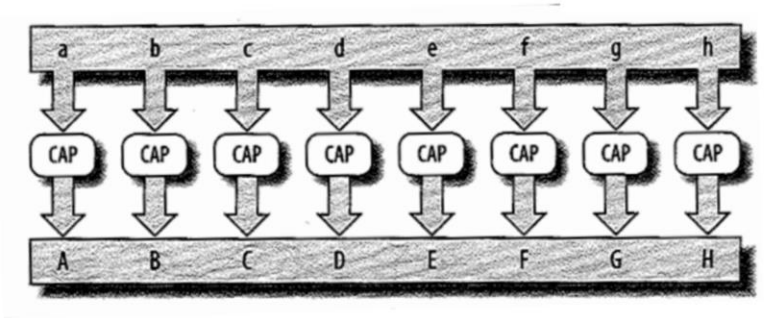
Task parallelism



- Several functions on the same data
- No dependencies between tasks, all can run in parallel

An example of task parallelism is shown here. Several functions are calculated on the same data: average, minimum, binary or geometric mean.

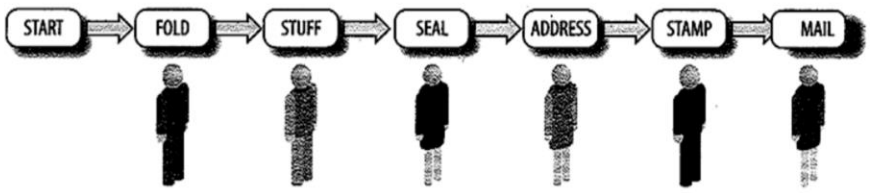
Data parallelism



- Can divide parts of the data between different tasks and perform the tasks in parallel
- Key: no dependencies between the tasks that cause their results to be ordered

This is an example of data parallelism: all characters in a text file must be converted to upper case.

Pipeline parallelism

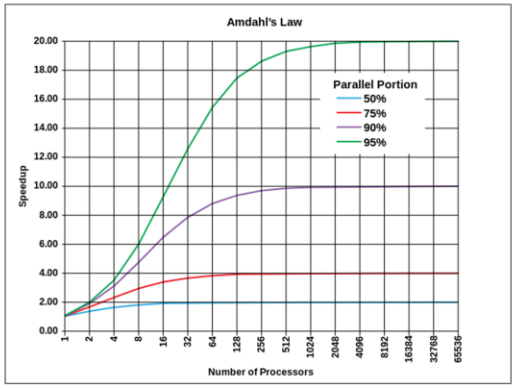


- Output of one task is the input to the next one
- Each task can run in parallel
- Throughput impacted by the longest-latency element in the pipeline

Amdahl's law

Determines the maximum speed-up *S* by dividing an amount of work with parallel fraction *P* over *N* units

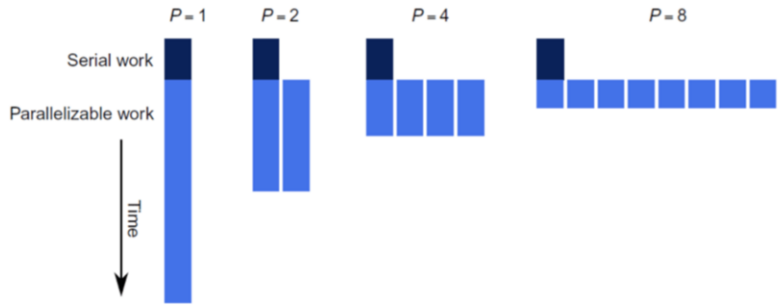
$$S \leq \frac{1}{(1-P)+P/N}$$



Speedup is limited by the time needed for the sequential fraction of the program

Amdahl's law prescribes the upper limit on the speedup that can be achieved by dividing a program (work load) over N processors.
Each program can be divided into a fraction *P* that is parallelizable and a portion *(1-P)* that is intrinsically serial. The time needed to execute the serial portions is unaffected by the number of parallel processing units. The law gives an upper bound that can only be realized by ignoring the overhead due to message passing, gathering of results, etc...

Philosophical question

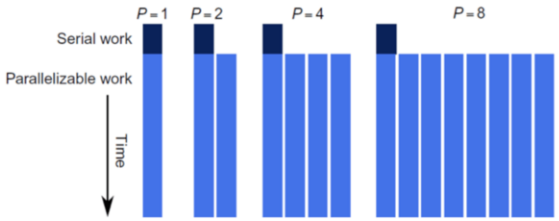


If the maximum speed-up is bounded by Amdahl's law, then why are we building supercomputers with 1000s of nodes?

Gustafson's law

- As processor power increases, programmer's tend to increase the size of the problem
  - set the problems to be solved within a practical fixed time
  - e.g. more pixels, larger scale, smaller time steps...
- Key assumption: total amount of work varies linearly with the number of processors

$$T_{seq} = a + b \cdot N$$



Gustafson's law

$$S = \frac{T_{seq}}{T_{par}} = \frac{a+b \cdot N}{a+b}$$

$$S = N - \alpha \cdot (N-1) \qquad \alpha = a/(a+b)$$

- α = sequential fraction of the total execution time on a parallel session
- if α is small, then the speed-up is almost linear with the amount of processors

Gustafson's law indicates that when the available computational power increases with a factor N, you can handle workloads that are similarly scaled while keeping the total time constant.

The difference between Amdahl's law and Gustafson's law lies in whether you want to make a program run faster with the same workload or run in the same time with a larger workload. History clearly favors programs attacking and solving larger, more complex problems, so Gustafson's observations fit the historical trend. Nevertheless, Amdahl's Law still haunts you when you need to make an application run faster on the same workload to meet some latency target.

Bibliography: parallel computing

- D. Ernst, Introduction to Accelerators and GPGPU
- D. Negrut, High Performance Computing for Engineering Applications, http://sbel.wisc.edu/Courses/ME964/2013/Lectures/lecture0918.pdf
- B. Barney, Introduction to Parallel Computing https://computing.llnl.gov/tutorials/parallel_comp
- http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concurrency-basics.pdf
- M. Gillespie, Amdahl's Law, Gustafson's Trend, and the Performance Limits of Parallel Applications
- http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980?pgno=2

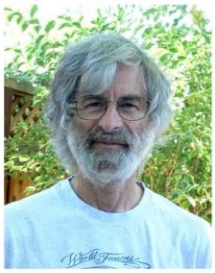
Outline

- Parallel computing
- Distributed computing
  - what is distributed computing
  - CAP theorem and its extensions
  - RAFT consensus algorithm

Distributed system

"A distributed system is one where I can't get the job done because a computer I never heard of drashed"

Leslie Lamport



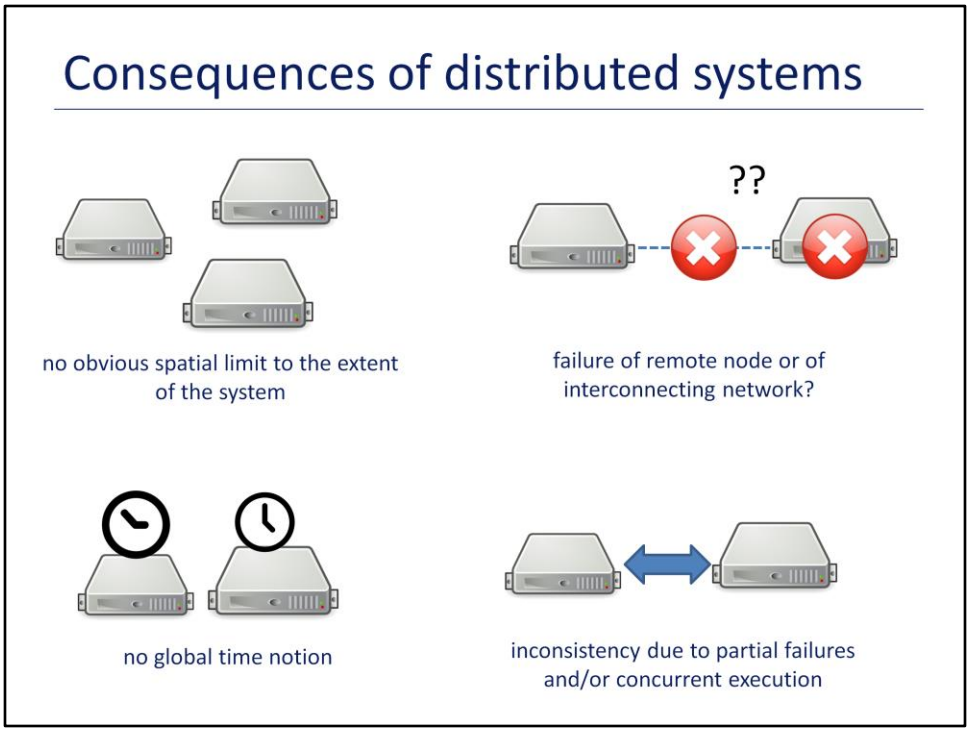
Distributed system

Distributed system

System where hardware and software components are located at networked computers. Components communicate and coordinate their actions only by passing messages.

"The network is the computer"

WWW, grids,
P2P, cloud computing



There are a number of natural consequences of distributed systems.

1) There is no spatial limit on the number of nodes in the distributed systems. Nodes of a single distributed system can be located on the same server, on different servers in the same datacenters, or on servers in different locations.
2) When communication with a remote node fails, it is hard for the requesting node to decide if the failure is the consequence of a network disruption or a crash of the node itself.
3) Each node of the distributed system has its own clock. This means you can not use locally generated timestamps for ordering distributed events.
4) Components are likely to execute concurrently. Partial failures are likely to happen, while the others continue to work. This results in inconsistent views and inconsistencies between the nodes.

Fallacies of distributed computing

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology does not change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

There is no such thing as a free lunch.

The fallacies of distributed computing are a set of assumptions that programmers new to distributed applications invariably make. These assumptions ultimately prove false, resulting either in the failure of the system, a substantial reduction in system scope, or in large, unplanned expenses required to redesign the system to meet its original goals.

1. The network is reliable – Hardware failures can never be excluded: power failures, someone tripping over a network cord, etc… On the infrastructure side, this means you need to think about hardware redundancy and weigh the risks of failure vs. the required investment (e.g. buying a spare switch). On the software side, you need to think about messages/calls getting lost. For one, you can use full reliable messaging.

2. Latency is zero – Latency can be relatively good on a LAN but deteriorates quickly when you move to WAN scenarios or internet scenarios. Even when you work on a LAN with Gigabit Ethernet you should bear in mind that the latency is much bigger then accessing local memory. Assuming the latency is zero you can be easily tempted to assume making a call over the wire is almost like making a local call. This mean you should strive to make as few as possible calls and assume you have enough bandwidth to move data in/out in these calls.
3. Bandwidth is infinite – Although network bandwidth is continuously growing, so does the amount of information we try to squeeze through it. Also, in the WAN,

packet loss and latency will limit your throughput (e.g. over TCP). While the previous fallacy stimulates to send fewer, but larger messages, this fallacy states that you should strive to limit the size of individual messages as well.

4. Network is secure – Build security in your software from Day 1. Security is usually a multi-layered solution that is handled on the network, infrastructure and application levels.
5. Topology does not change – Usually, the network topology is out of your control. The cloud infrastructure provider may add and remove servers or make other changes to the network. Try not to depend on specific endpoints or routes. Provide location transparency (e.g. multicast), discovery services and abstract the physical structure of the network (e.g. using DSN names instead of IP addresses)
6. There is one administrator – Your company might collaborate with external entities (e.g. hosting provider), your application might consume external services, etc… The DevOps strategy can help, because system adminstrators then become part of the development team. But with external parties, this is not always possible.
7. Transport cost is zero – Marshaling (serializing) information across the different network stack layers takes computer resources and adds to the latency. The second way to interpret this statement is that the monetary cost for running a network is not free.
8. The network is homogeneous – Most networks are not homogeneous. Even in a home network, you might encounter Linux devices, Windows PCs, NAS servers and mobile devices. At the network layer, this fallacy does not cause too much trouble since every device will talk IP. At the application level, you have to assume interoperability will be needed sooner or later. Do not rely on proprietary protocols and use standard technologies.

Why distributed systems?

- Many challenges
  - difficult to manage
  - no global time notion
  - no global state
  - almost always concurrent execution
  - (partial) failures ~~likely to~~ will to happen
- Many advantages
  - resource sharing
    - information (e.g. music files, group documents)
    - hardware (e.g. collective storage system, cloud infrastructure)
  - scalability: "easy" to adapt to larger user base
  - fault tolerance: "easy" to cope with failures

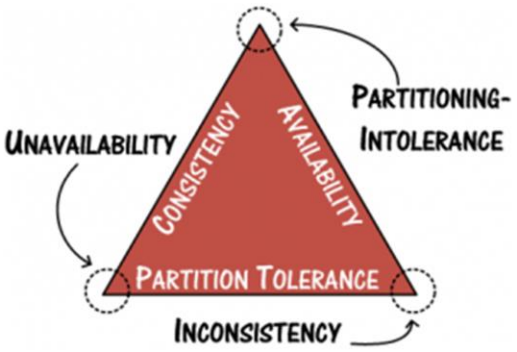
Despite the intrinsic difficulties you will have to cope with in a distributed system, there are many very important advantages.

- It allows for resource sharing: you can access information and/or services provided by 3rd parties
- It allows for scalability: if user demand increases, you can add more servers to distribute computation and data
- It allows for redundancy: replicating data or computation on multiple nodes allows that the overall system stays alive, even during failures.

Outline

- Parallel computing
- Distributed computing
  - what is distributed computing
  - CAP theorem and its extensions
    - Definition
    - Coping with CAP in real-world applications
  - RAFT consensus algorithm

CAP Theorem



© StackExchange

- describes trade-offs involved in distributed system
- impossible to provide the following *three guarantees at the same time*
  - **C**onsistency: all nodes see the same data at the same time
  - **A**vailability: all non-failing nodes are available for queries
  - **P**artition-tolerance: underlying system can be split in non-communicating groups

The CAP theorem was conjectured by Prof. Eric Brewer during a keynote talk at the PODC (Principle of Distributed Computing) conference in 2000. In 2002, other authors published a formal proof of Brewer's conjecture, rendering it into a theorem.

The theorem states that it is impossible for a distributed computer system to provide all three of the following guarantees:

- Consistency: all nodes see the same data at the same time. In other words, each server gives the correct response to each request
- Availability : a guarantee that every request receives a response about whether it succeeded or failed. Alternative formulations: every request received by a non-failing node in the system must result in a response; all (non-failing) nodes are available for queries)
- Partition tolerance: this refers to the underling system and not to the service. Servers partitioned to groups that are not able to communicate. The system continues to operate despite arbitrary partitioning due to network failures, at least one partition should remain functioning and accessible to the clients.

CAP Theorem

Distributed system = a collection of interconnected nodes that share data

Consistency

All nodes see the same data at the same time

Availability

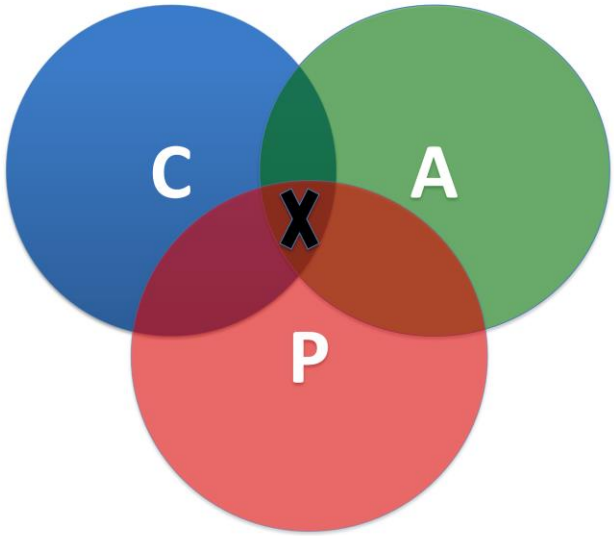
Every request gets a response on success/failure, regardless of the state of any individual node

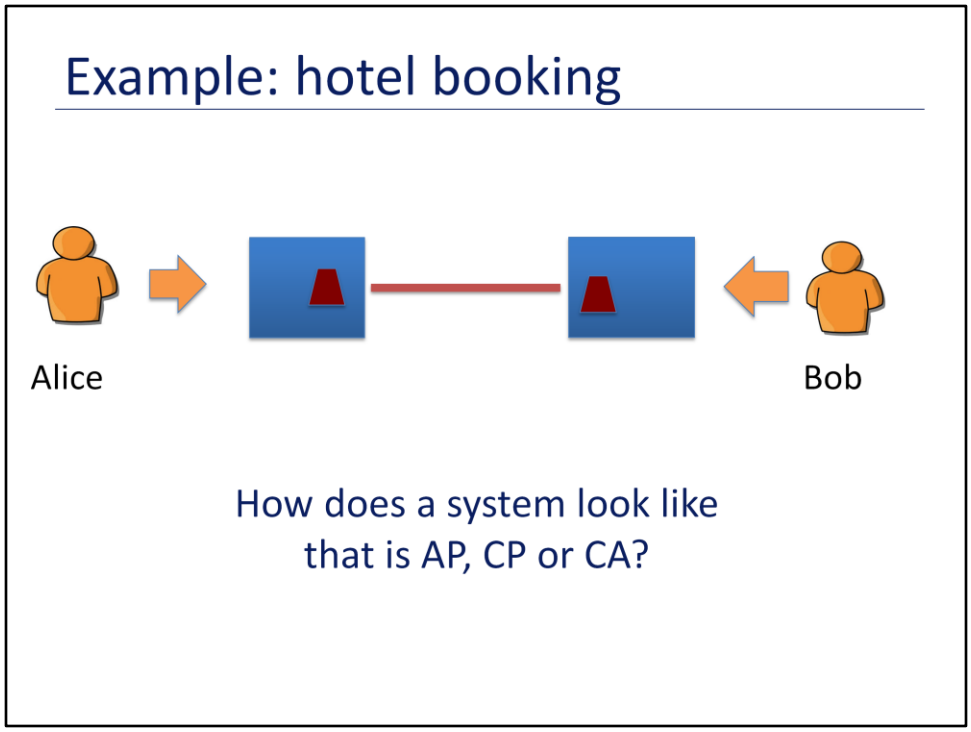
Partition Tolerance

The system continues to function despite message loss between nodes

This is a more visual view on the three elements in the CAP theorem.
A distributed system can satisfy any two of these guarantees at the same time **but not all three**.

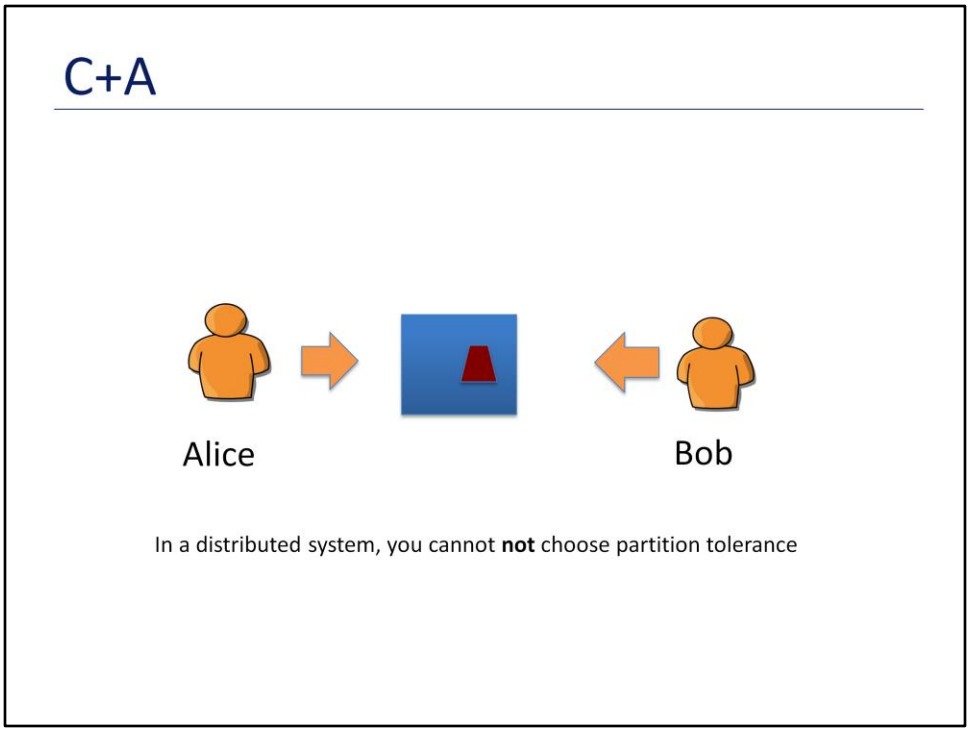
CAP theorem



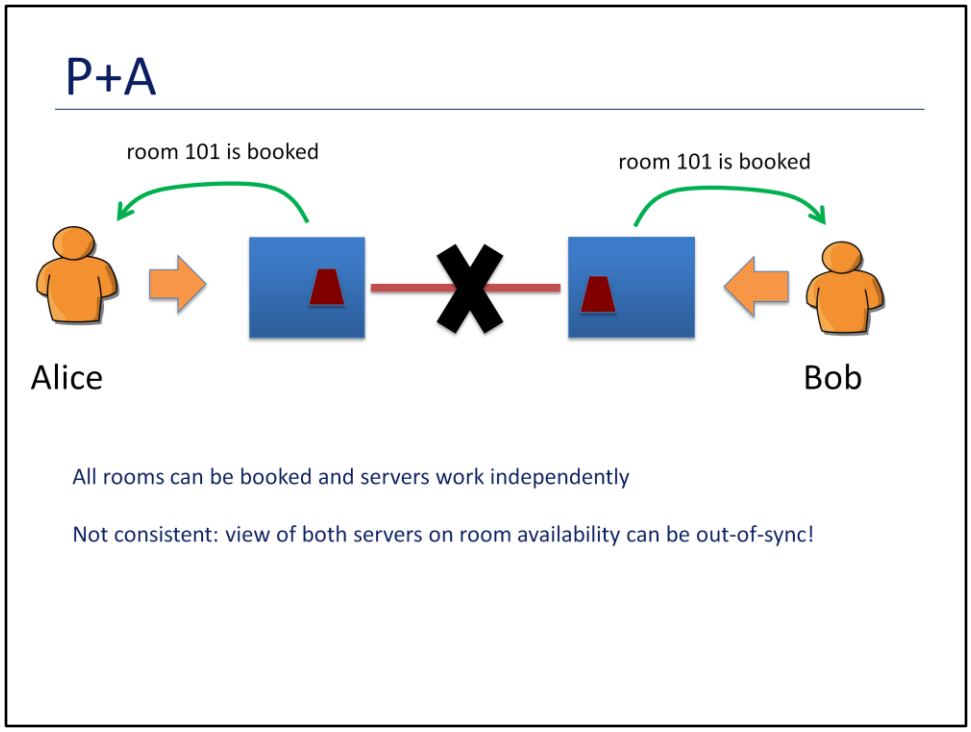


We will use a hotel booking system as a running example to illustrate the CAP theorem. The hotel booking system comprises two servers at physically different locations. Alice and Bob are simultaneously trying to book a room in a hotel. Of course, we want to avoid that a room is double-booked.

(note: the “last” hotel room is simply a metaphor for a specific data item that is read/writable by multiple users)

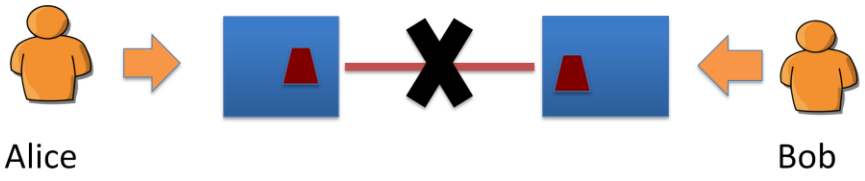


CA is only possible as a monolithic, single server database. As soon as you have two or more servers, you introduce a network link and you basically *require* "partition tolerance". In a distributed system, you cannot **not** choose partition tolerance.



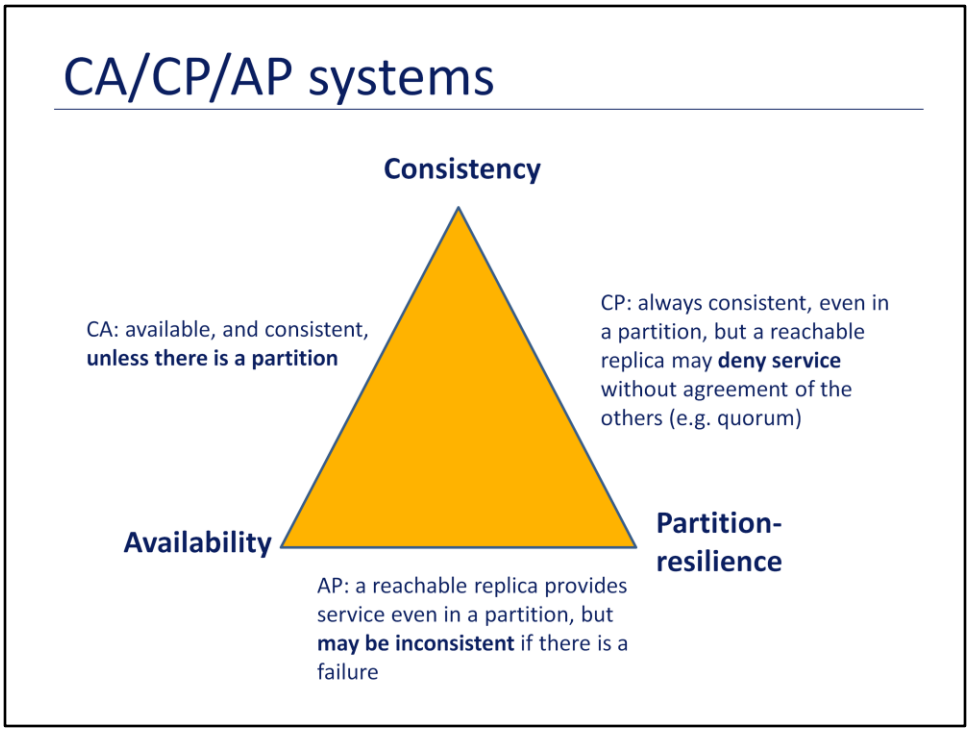
As soon as we have more than 1 server, we tolerate partitions. According to the CAP theorem, this means we must choose between A and C. In a P+A model, the system keeps functioning even during a network partition, or when a node fails. Clearly, consistency is sacrificed since a user can book a room on one server without the other server knowing this.

P+C



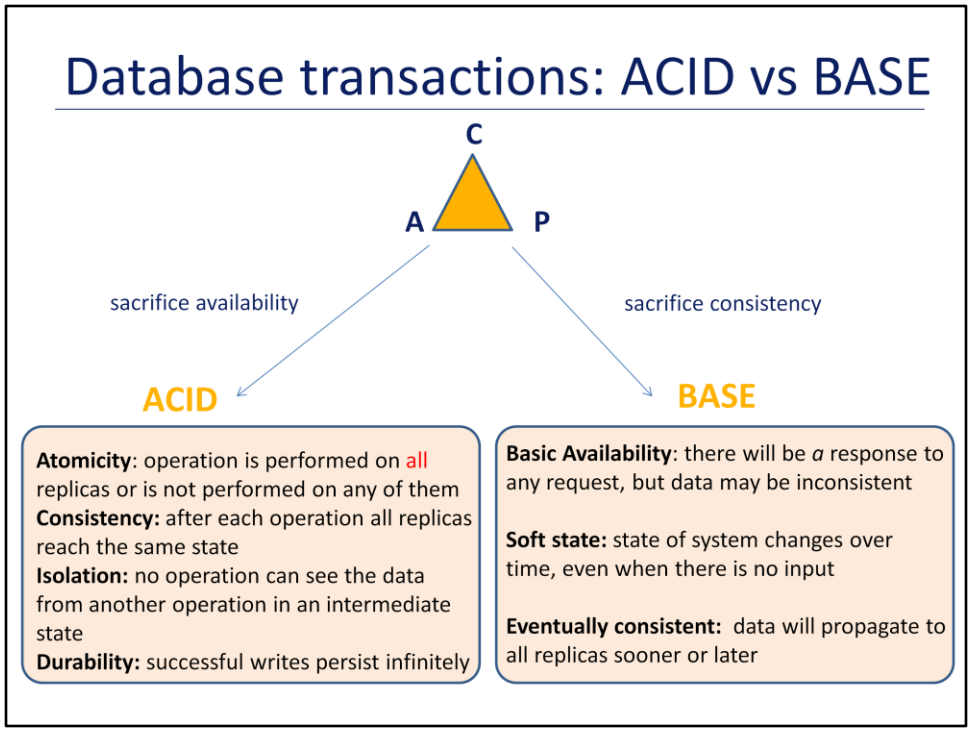
If we require consistency, then there is no other option than take at least one of the two servers offline during a partition, although all servers are working properly

If we don't allow the inconsistency of the previous slide, then we must take at least one of the two servers offline during network partitions.



In practice, CAP takes place during a timeout. Then a decision should be made:

- Cancel the operation and thus decrease availability
- Continue the operation and be prone to inconsistency



Any realistic data scaling system will have to scale, either horizontally (sharding, replicating same data over multiple nodes) or vertically (splitting data according to their function, e.g. storing user data on one node, product information on another, etc.). Of course, horizontal and vertical scaling can (will) be combined, but in any case we end up with a distributed database system that is subject to the CAP theorem. This means that when a partition occurs, you have to choose between availability and consistency when starting a new transaction with the database.

If we choose consistency, then *database transactions* follow the ACID pattern:
Atomicity. All of the operations in the transaction will complete, or none will.
Consistency. The database will be in a consistent state when the transaction begins and ends.
Isolation. The transaction will behave as if it is the only operation being performed upon the database.
Durability. Upon completion of the transaction, the operation will not be reversed.

Database vendors introduced a technique known as 2PC (two-phase commit) for providing ACID guarantees across multiple database instances. The protocol is broken into two phases:

- First, the transaction coordinator asks each database involved to precommit the operation and indicate whether commit is possible. If all databases agree the commit can proceed, then phase 2 begins.

- The transaction coordinator asks each database to commit the data.

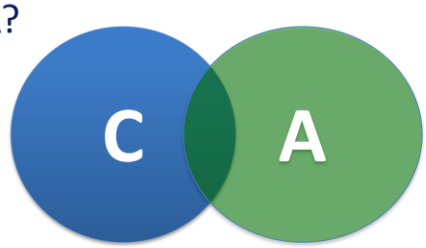
If any database vetoes the commit, then all databases are asked to roll back their portions of the transaction. This way, we are getting consistency across partitions. But if one of the nodes goes down, any transaction will fail. Essentially, a transaction involving two database nodes in a 2PC commit will have the availability of the product of the availability of each database. For example, if we assume each database has 99.9 percent availability, then the availability of the transaction becomes 99.8 percent, or an additional downtime of 43 minutes per month.

On the other hand, if we sacrifice consistency and prefer availability, the processing of transactions follows the BASE acronym:
Basically Available: This constraint states that the system does guarantee the availability of the data as specified by the CAP Theorem -- there will be a response to any request. But, that response could still be a 'failure' to obtain the requested data or the data may be in an inconsistent or changing state.
Soft state: The state of the system could change over time, so even during times without input there may be changes going on due to 'eventual consistency,' thus the state of the system is always 'soft.'
Eventual consistency: The system will *eventually* become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one.

COPING WITH CAP
IN REAL-WORLD SYSTEMS

A popular misconception: 2 out 3

- How about distributed CA?
- Can a distributed system (with unreliable network) really be not tolerant of partitions?



CAP Theorem 12 year later

- Prof. Eric Brewer: father of CAP theorem
  - "The "2 of 3" formulation was always **misleading** because it tended to oversimplify the tensions among properties. ...
  - **CAP prohibits only a tiny part of the design space**: *perfect availability and consistency in the presence of partitions*, which are rare."

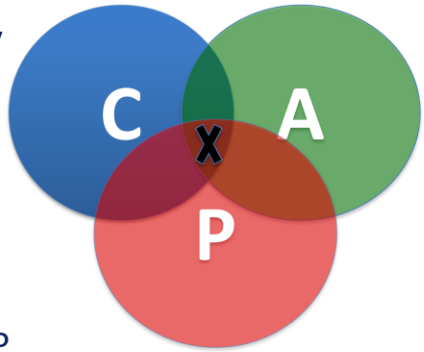
http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

The CAP theorem has been victim of its own success and is subject to some misconceptions:

- The popular belief is that CAP means *pick any two*. The confusion is about the existence of CA systems, which pretend that partition tolerance is optional, or claim that partitions don't happen. In reality, you can't sacrifice partition tolerance because partitions happen in real large-scale systems all the time.
- The second misconception is that the CAP theorem means *you can't be consistent and available during partitions*. That's not true. Specifically, the CAP theorem only prevents *everybody* from being consistent and available, not *anybody* (some literature calls this *always available*). It doesn't prevent clients and replicas on the majority side of simple partitions from making progress, and experiencing both consistency and availability. There are other restrictions on this, but they aren't CAP restrictions.
- The third misconception is that the *consistency* in CAP is all or nothing, and that you can't offer any consistency guarantees at all during partitions. In reality, many very useful consistency models can be offered on all sides of a partition. Implementation tricks like session stickiness and client-side caching can allow systems to offer useful models like *read your writes*, *monotonic reads* and even *causal consistency*.

Consistency or Availability

- Consistency and Availability is not "binary" decision
- AP systems relax consistency in favor of availability – but are not inconsistent
- CP systems sacrifice availability for consistency- but are not unavailable
- This suggests both AP and CP systems can offer a degree of consistency, and availability, as well as partition tolerance



The practical implications of the CAP theorem are not as stringent as they appear, because in practical situations you are often allowed to relax (but not give up completely) consistency or availability.

Instead of actually having to choose between a CP or AP system, the important message of the CAP theorem is that you have to balance between Consistency with Availability.

CP: Best Effort Availability

- guarantees consistency, regardless of network behavior
- when communication is typically reliable
  - E.g. servers in same datacenter, partitions are rare (but not impossible)
- Example:
  - Majority protocols
  - Distributed Locking (Google Chubby Lock service)
- Trait:
  - Pessimistic locking
  - Make minority partition unavailable

AP: Best Effort Consistency

- Sometimes being unavailable is not an option
- Inconsistency is not a major problem
  - Best effort for up-to-date data
  - No assurance that all users get the same content
- Example:
  - Web Caching
  - DNS
- Trait:
  - Optimistic
  - Expiration/Time-to-live
  - Conflict resolution

Types of Consistency

- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.
- Weak Consistency
  - It is **not guaranteed** that subsequent accesses will return the updated value.
- **Eventual Consistency**
  - Specific form of weak consistency
  - It is guaranteed that if **no new updates** are made to object, **eventually** all accesses will return the same updated value (e.g., *propagate updates to replicas in a lazy fashion*)
  - in principle, does not impose global ordering

Eventual consistency is a consistency model, which is used in many large distributed databases. Such databases require that all changes to a replicated piece of data eventually reach all affected replicas. The storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value.

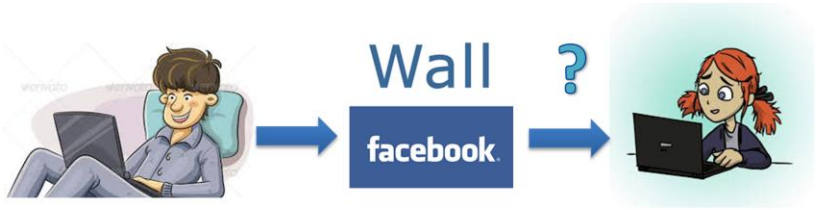
Consider a case where data item R=0 on all three nodes. Assume that we have the following sequence of writes and commits: W(R=3) C W(R=5) C W(R=7) C in node 0. Now read on node 1 could return R=5 and read from node 2 could return R=7. This is eventually consistent as long as eventually read from all nodes return the same value. Note that this final value could be R=5. **Eventual consistency does not restrict the order in which the writes must be executed.**

Definition of Eventual consistency:
* *Eventual delivery*: An update executed at one node eventually executes at all nodes.
* *Termination*: All update executions terminate.
* *Convergence*: Nodes that have executed the same updates eventually reach an equivalent state (and stay).

Eventual Consistency: Facebook

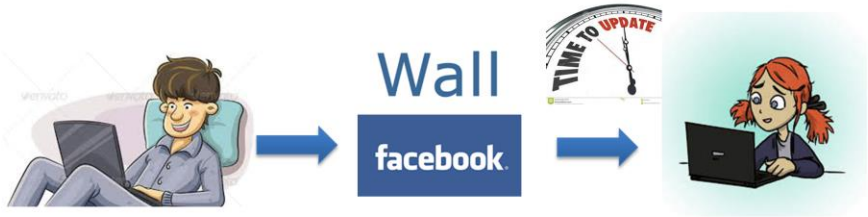
- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but finds:

  - **Nothing is there!**



Eventual Consistency: Facebook

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back:
  - **She finds the story Bob shared with her!**



The reason that Alice doesn't see the post immediately is because Facebook applies an eventual consistent model. Facebook has more than 1 billion active users, so it's non-trivial to efficiently and reliably store the huge amount of data generated at any given time. An eventual consistent model offers the option to reduce the load and improve availability.

Eventual Consistency: Facebook

- Reason: it is possible because Facebook uses an **eventual consistent model**
- Why Facebook chooses eventual consistent model over the strong consistent one?
  - Facebook has more than 1 billion active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to **reduce the load and improve availability**

Eventual Consistency: Dropbox

- Dropbox enabled immediate consistency via synchronization in many cases.
- However, what happens in case of a network partition?

Eventual Consistency: Dropbox

- Let's do a simple thought experiment here:
  - Open a file in your Dropbox
  - Disable your network connection (e.g., Wi-Fi, 4G)
  - Try to edit the file in the Dropbox: can you do that?
  - Re-enable your network connection: what happens to your Dropbox folder?

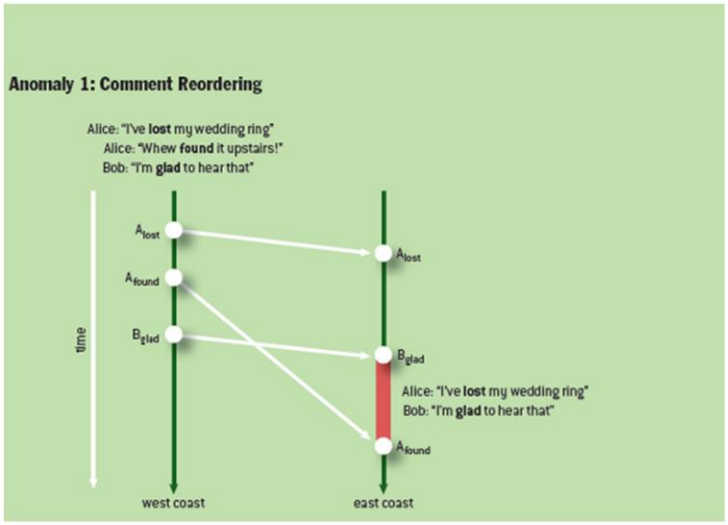
Also Dropbox embraces eventual consistency. Immediate consistency is impossible in case of a network partition. But even when online, it is not desirable to have immediate (strong) consistency: users will feel bad if their Word documents freeze each time they save it, simply due to the large latency to update all devices across WAN.

Dropbox is oriented to **personal syncing** and not on collaboration. So having eventual consistency is not a real limitation.

Eventual Consistency: ATM

- In design of automated teller machine (ATM):
  - Strong consistency appear to be a nature choice
  - However, in practice, **A beats C**
  - Higher availability means **higher revenue**
  - ATM will allow you to withdraw money *even if the machine is partitioned from the network*
  - However, it puts **a limit** on the amount of withdraw (e.g., $200)
  - The bank might also charge you a fee when a overdraft happens

Eventual consistency goes without strict ordering



Eventual Consistency Variations

In practice, clients of the distributed data storage system get better guarantees than *pure* EC

- Read-your-write consistency
  - A process always accesses the data item after it's update operation and never sees an older value
- Session consistency
  - As long as session exists, system guarantees read-your-write consistency
  - Guarantees do not overlap sessions

Read Your Writes Consistency (RYWC) guarantees that a client that has written a version *n* will thereafter always be able to read a version that is at least as new as *n*. This helps, for example, to avoid user irritation when a person checks his bank account statement after he has wired some money to another person, but does not see his account credited. If there is no RYWC, he might think that the transfer was unsuccessful and wire the same amount of money again. Generally, RYWC avoids situations where a user or application issues the same request several times because it gets the impression that the request failed the first time. For idempotent operations reissuing requests causes only additional load on the system, while reissuing other requests create severe inconsistencies.

Eventual Consistency Variations

- Monotonic read consistency
  - If a process has seen a particular value of data item, any subsequent access by that process will never return any previous values
- Monotonic write consistency
  - The system guarantees to serialize the writes by the *same* process
- In practice
  - A number of these properties can be combined
  - Monotonic reads and read-your-writes are most desirable

Monotonic Read Consistency guarantees that a client that has read a version *n* will thereafter always read versions >= n. This is helpful as from an application perspective data visibility might not be instantaneous but versions come at least in chronological order: the system never "goes backward" in time.

Monotonic Write consistency guarantees that two updates by the same client will be serialized in the order that they arrive at the storage system. This is useful to avoid seemingly lost updates when an application first writes and then updates a datum (data entry) but the update is executed before the initial write and is, thus, overwritten.

Dynamic Tradeoff between **C** and **A**

Many applications require neither strong consistency nor continual availability.

Applications specify level of continuous consistency: e.g. airline reservation system

- Many free seats – sacrifice consistency
- A few places left – sacrifice availability

When designing your system, CAP only describes that you have to balance (and not choose) between consistency and availability. The optimal trade-off depends on the application at hand and even inside a single application the desired trade-off might vary during execution.

Consider the example of an airline reservation. When most of the seats of a specific flight are available, it is ok to rely on somewhat out-of-date data. Availability is more critical, since being unavailability might mean that customers shift to another airline. However, when the plane is close to be filled, the reservation system needs more accurate data to ensure the plane is not overbooked. In this case consistency is more critical, because the fees that you have to pay to passengers with denied boarding is too high to compensate for the risk of losing a few customers when being unavailable.

Heterogeneity: Segmenting **C** and **A**

- No single uniform requirement for entire system
  - Some aspects require strong consistency
  - Others require high availability
- Segment your system into different components
  - Each provides different types of guarantees
- Overall guarantees neither consistency nor availability
  - But each part of the service gets exactly what it needs
- Can be partitioned along different dimensions

The trade-off between C and A should (and most likely will) not be the same for all the components in your system. When designing your system, you should segment it into different components that each receive exactly those guarantees they need.

Discussion

- In an e-commercial system (e.g., Amazon, e-Bay, etc), what are the trade-offs between consistency and availability you can think of? What is your strategy?
- Hint -> Things you might want to consider:
  - Different types of data (e.g., shopping cart, billing, product, etc.)
  - Different types of operations (e.g., query, purchase, etc.)
  - Different types of services (e.g., distributed lock, DNS, etc.)
  - Different groups of users (e.g., users in different geographic areas, etc.)

Examples of partitioning C/A

- Data Partitioning
- Operational Partitioning
- Functional Partitioning
- User Partitioning
- Hierarchical Partitioning

Partitioning Examples

Data Partitioning

- Different data may require different consistency and availability
- Example:
  - Shopping cart: high availability, responsive, can sometimes suffer anomalies
  - Product information need to be available, slight variation in inventory is sufferable
  - Checkout, billing, shipping records must be consistent

Partitioning Examples

Operational Partitioning

- Each operation may require different balance between consistency and availability
- Example:
  - Reads: high availability; e.g., “query”
  - Writes: high consistency, lock when writing; e.g., “purchase”

Partitioning Examples

Functional Partitioning

- System consists of sub-services
- Different sub-services provide different balances
- Example: A comprehensive distributed system
  - Distributed lock service (e.g., Chubby) :
    - Strong consistency
  - DNS service:
    - High availability

Partitioning Examples

User Partitioning

- Try to keep related data close together to assure better performance
- Example: Craigslist
  - Might want to divide its service into several data centers, e.g., east coast and west coast
    - Users get high performance (e.g., high availability and good consistency) if they query servers closest to them
    - Poorer performance if a New York user query Craigslist in San Francisco

Partitioning Examples

Hierarchical Partitioning

- Large global service with local “extensions”
- Different location in hierarchy may use different consistency
- Example:
  - Local servers (better connected) guarantee more consistency and availability
  - Global servers has more partition and relax one of the requirement

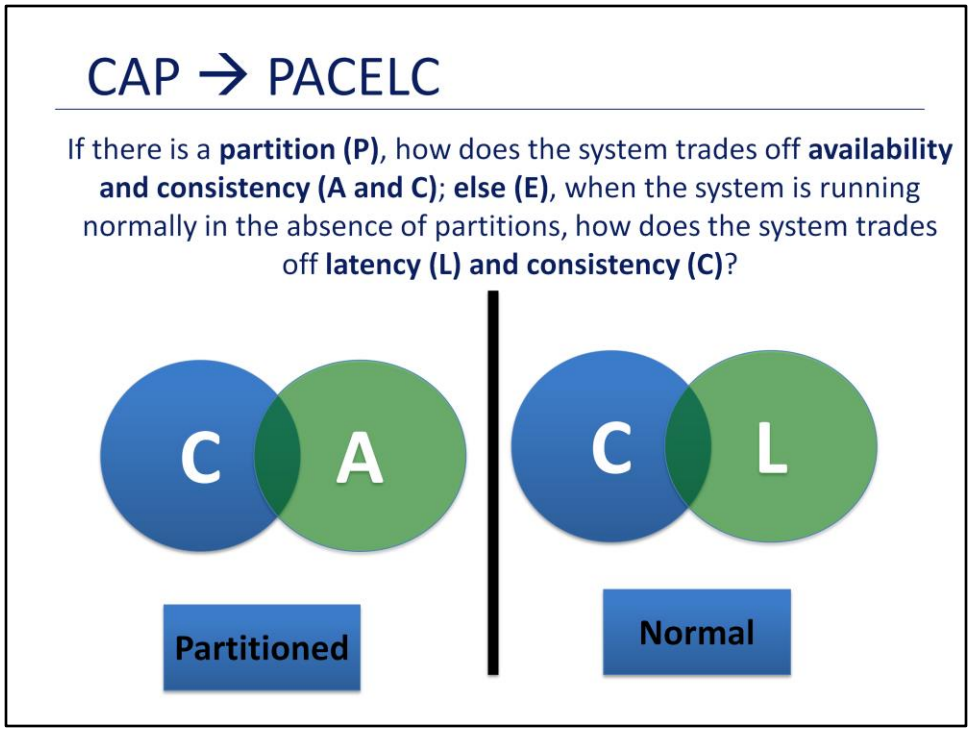
What if there are no partitions?

The occurrence of failure causes CAP tradeoffs

The possibility of failure results in Consistency/Latency tradeoffs

- Availability ~ Latency
  - intuition: unavailable system provides extreme latency
- High availability → need to replicate → consistency problem

Availability and latency are arguably the same thing: an unavailable system provides extreme latency. Latency exists even without network partitions. But on the other hand, when a system runs long enough, at least one component will fail. Highly available systems will thus need to replicate data, which in turns causes challenges to guarantee consistency.



The acronym PACELC makes clear that even in the **absence** of partitions there's a tradeoff between consistency and latency. In the general case, though not all cases, consistency requires a level of coordination which prevents systems from being always available, and increases latency when no partition is present. The matter of latency, which is of great practical importance in real-world systems, isn't captured at all in CAP.

A more complete description of the space of potential tradeoffs for distributed system is thus given by the PACELC acronym. PACELC stands for the following sentence: "If there is a **partition (P)**, how does the system trade off **availability and consistency (A and C)**; **else (E)**, when the system is running normally in the absence of partitions, how does the system trade off **latency (L) and consistency (C)**?"

This sentence (and the acronym) are written in a seminal article of Daniel Abadi on consistency in modern distributed database system design:

Abadi, Daniel J. "Consistency tradeoffs in modern distributed database system design." Computer-IEEE Computer Magazine 45.2 (2012): 37.

Examples

- **PC/EC Systems:** Refuse to give up consistency and pay the cost of availability and latency
  - BigTable, Hbase, VoltDB/H-Store
- **PA/EL Systems:** Give up both Cs for availability and lower latency
  - Dynamo, Cassandra, Riak
- **PA/EC Systems:** Give up consistency when a partition happens and keep consistency in normal operations
  - MongoDB
- **PC/EL System:** Keep consistency if a partition occurs but gives up consistency for latency in normal operations
  - Yahoo! PNUTS

The first two categories of PACELC are very clear. PC/EC is the most consistent class of systems, which never give up consistency. PA/EL systems don't try hard to be consistent, and rather take the opportunity to reduce latency and gain availability by reducing coordination.

The trickier ground starts with PA/EC. These types of systems give up consistency when there is a partition, and are consistent when there isn't. That's more subtle than it looks. When is there a partition? How long does the network need to be down before there is a partition? Is a single dropped connection or lost packet a partition? That may seem like nit picking, but there's an important line to be drawn between *partition* and *not partition*. PACELC doesn't help there.

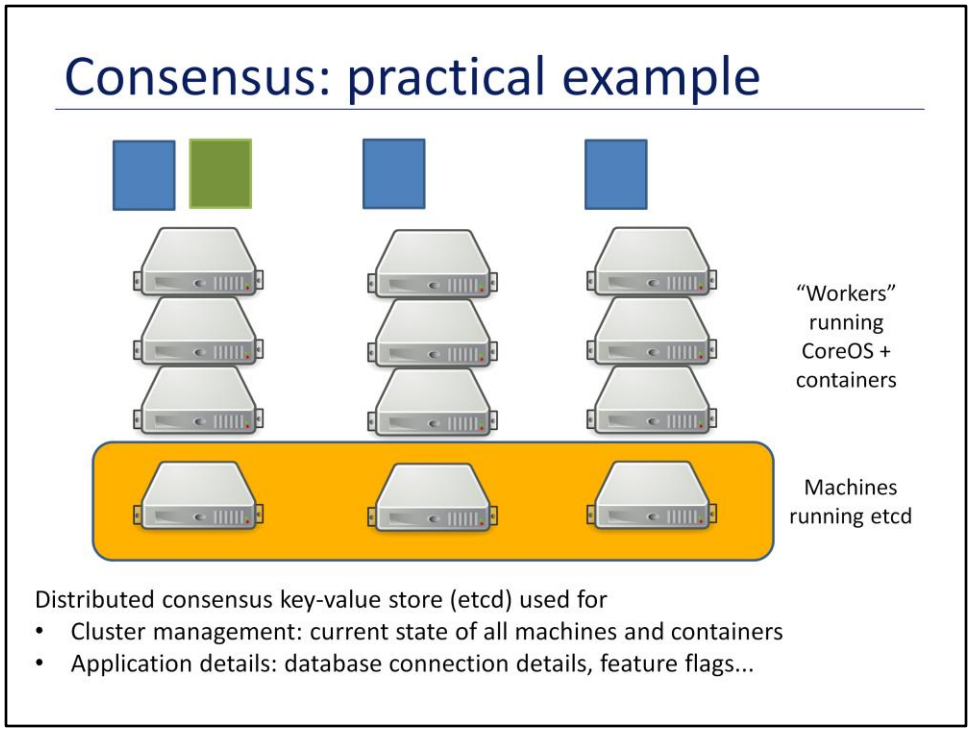
If PA/EC is tricky, PC/EL is madness. What does it mean to be more consistent during a partition? Daniel Abadi (who coined PACELC) says that is the wrong question: Yahoo! PNUTS is a PC/EL system. In normal operation, it gives up consistency for latency; however, if a partition occurs, it trades availability for consistency. This is admittedly somewhat confusing: according to PACELC, PNUTS appears to get more consistent upon a network partition. However, PC/EL should not be interpreted in this way. PC does not indicate that the system is fully consistent; rather it indicates that the system does not reduce consistency beyond the baseline consistency level when a network partition occurs—instead, it reduces availability.

Bibliography: distributed computing

- D. Wang, Cloud Computing, CSE 40822, http://www3.nd.edu/~dthain/
- M. Brooker, CAP and PACELC: Thinking More Clearly About Consistency, http://brooker.co.za/blog/2014/07/16/pacelc.html
- I. Tsalouchidou, The Cap theorem in depth, http://www.slideshare.net/ioanna_tsalouchidou/cap-in-depth
- D. Bermbach, Benchmarking, Consistency, Distributed Database Management Systems
- D. J. Abadi, Consistency Tradeoffs in Modern Distributed Database System Design
- D. Power, What is ACID and BASE in database theory? http://dssresources.com/faq/index.php?action=artikel&id=281

Outline

- Parallel computing
- Distributed computing
  - what is distributed computing
  - CAP theorem and its extensions
  - RAFT consensus algorithm
    - definition
    - challenges
    - practical example: etcd



Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. A typical example where consensus algorithms are used is in a server cluster running containers. The set-up consists of worker nodes, complemented with 3-5 machines running central services (including the distributed key-value store). Each of the workers will use the distributed key-value store on the central machines via local proxies.

An example central service is the fleet manager. This fleet manager handles the scheduling of containers across the cluster machines and keeps them running even if the original host they were running on is terminated. The fleet manager needs a shared view of the current state of all the machines and containers running in the cluster. This functionality is provided by e.g. etcd, a distributed, consistent key-value store used for storing shared configuration and information in a cluster.

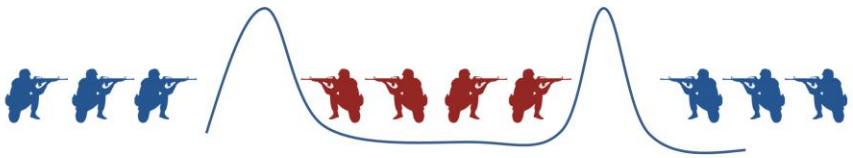
Also applications can read and write data into the key-value store. A simple use-case is to store database connection details or feature flags in etcd as key value pairs. These values can be watched, allowing your app to reconfigure itself when they change. Advanced users take advantage of the consistency guarantees to implement database master elections or do distributed locking across a cluster of workers.

Requirements for consensus

- **Agreement**
  - All correct processes must agree on the same value
- **Termination**
  - All processes must eventually decide on an output value
- **Validity**
  - If all correct processes propose the same value *v*, then all correct processes decide *v*
- **Integrity**
  - If a correct process decides *v*, then *v* must have been processed by some correct process

The idea behind integrity is that we want to exclude trivial solutions that just decide 'No' whatever the initial set of values is. Such an algorithm would satisfy termination and agreement, but would be completely vacuous, and no use to use at all.

Consensus

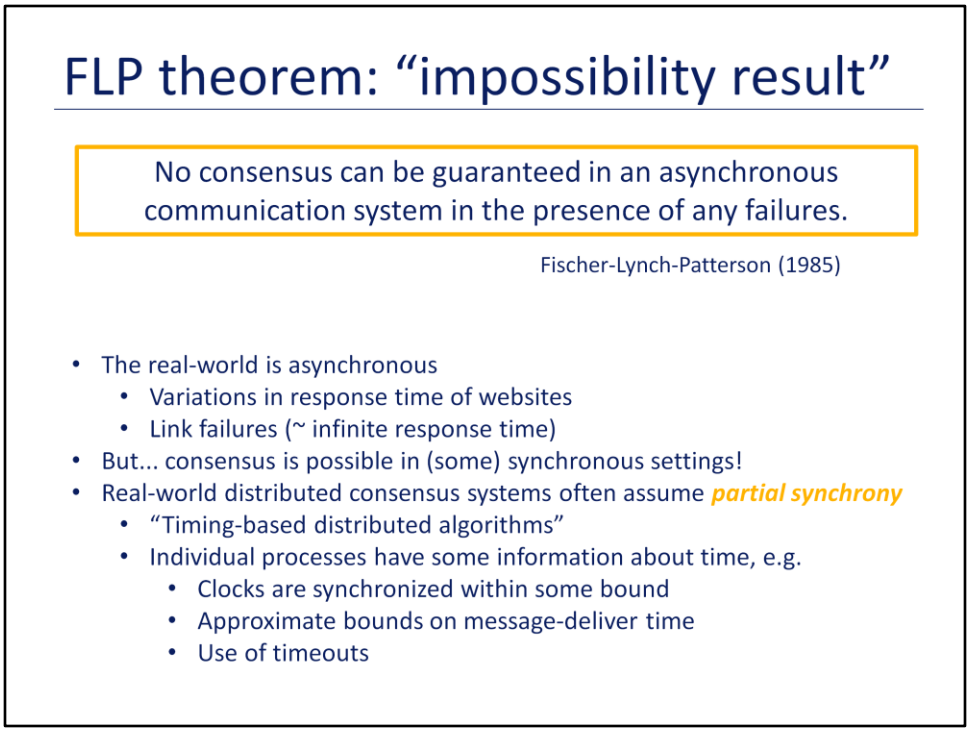


- generals of left and right blue army must decide on value of attack (= 0 or 1)
  - blue army only wins if both sides attack at the same time
  - to reach consensus, they must send messages
  - but these messages may get lost during transmission
- impossible to reach consensus!
  - my message may be intercepted/lost
  - is the other army still there?
- main reason: asynchronous character of the messaging

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Some of the challenges in reaching consensus between distributed entities are exemplified in the well known "Two army problem". Suppose we have two allied (blue) armies that have encircled the red army. The red army can only be beaten when both blue armies attack simultaneously. So, to reach consensus, the general of the left blue army will have to communicate with the general of the right blue army.

However, using messaging, it is impossible to reach consensus on the decision to attack or not. The major problem is that the armies use asynchronous messaging: there is no upper boundary on the time in which one army may expect an answer from the other army. In other words: for the left army it is impossible to know why the other army has not responded: its own message could be lost, the general of the other army might still be thinking about his strategy, or the reply of the other army can be lost during transmission.

Intuitive proof of the above: assume protocol P is the shortest protocol (i.e. with a minimum number of messages) that solves the attack decision problem. Suppose now that the last message of protocol P does not reach its destination. Since protocol P is correct, consensus must be reached in any case. This means, the last message was useless, and then P could not be the shortest!



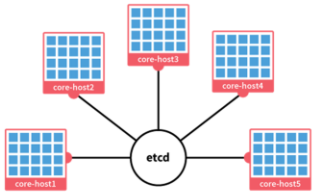
In 1985, Fischer, Lynch and Patterson formulated their “Impossibility Result” theorem which states that no consensus can be guaranteed in an asynchronous communication system in the presence of any failures. In an asynchronous timing assumption, there is no time bound on how long it takes for a message to be delivered.

Although the theorem has been proven mathematically, we provide here only an intuitive explanation. The asynchronous assumption makes it impossible to differentiate between failed and slow processes. A “failed” process may just be slow, or can rise from the dead at exactly the wrong time. Therefore, *termination* (liveness) cannot be guaranteed. On the other hand, an slow process may decide differently than other processes, thus violating the agreement property of consensus.

Because the real-world operates asynchronous, the FLP theorem is very important. It states that we must introduce at least a partial level of synchrony to be able to reach consensus. With partial synchrony, individual processes have some information about time, e.g. using clock synchronization within some bound, by setting approximate bounds on message-deliver time, by the use of timeouts...

RAFT algorithm

- New protocol (2014)
- Designed for *understandability*
- Used in *etcd*
- Before, PAXOS was the default standard
  - exceptionally difficult to understand



> "The dirty little secret of the NSDI community is that at most five people really, truly understand every part of PAXOS ☺"
> *anonymous NSDI reviewer*

  - very difficult to implement

> "There are significant gaps between the description of the PAXOS algorithm and the needs of a real-world system... The final system will be based on an unproven protocol"
> *authors of Chubby (distributed lock service from Google)*

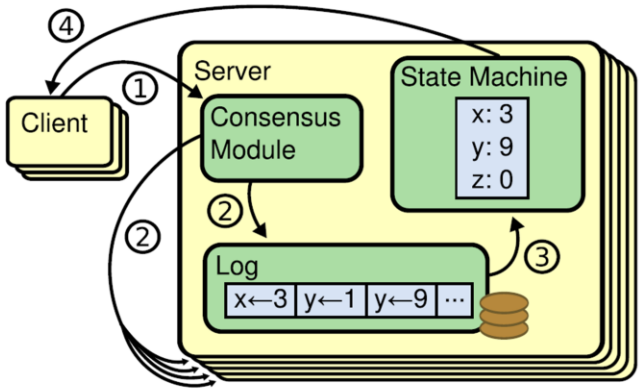
The Paxos consensus algorithm dominated the discussion of consensus algorithms over the last decade. Paxos first defines a protocol capable of reaching agreement on a single decision, such as a single replicated log entry. We refer to this subset as *single-decree Paxos*. Paxos then combines multiple instances of this protocol to facilitate a series of decisions such as a log (*multi-Paxos*). Unfortunately, PAXOS has two significant drawbacks.

The first drawback is that Paxos is exceptionally difficult to understand. The full (original) explanation is notoriously opaque; few people succeed in understanding it, and only with great effort. Paxos' opaqueness derives from its choice of the single-decree subset as its foundation. Single-decree Paxos is dense and subtle: it is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this, it is difficult to develop intuitions about why the single degree protocol works. The composition rules for multi-Paxos add significant additional complexity and subtlety.

The second problem with Paxos is that it does not provide a good foundation for building practical implementations. One reason is that there is no widely agreed upon algorithm for multi-Paxos. Furthermore, the Paxos architecture is a poor one for building practical systems; this is another consequence of the single-degree decomposition. For example, there is little benefit to choosing a collection of log entries independently and then melding them into a sequential log; this just adds

complexity. It is simpler and more efficient to design a system around a log, where new entries are appended sequentially in a constrained order. Another problem is that Paxos uses a symmetric peer-to-peer approach at its core. This makes sense in a simplified world where only one decision will be made, but few practical systems use this approach. If a series of decisions must be made, it is simpler and faster to first elect a leader, then have the leader coordinate the decisions. As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, discovers the difficulties in implementing it, and then develops a significantly different architecture.

State machines & the distributed log



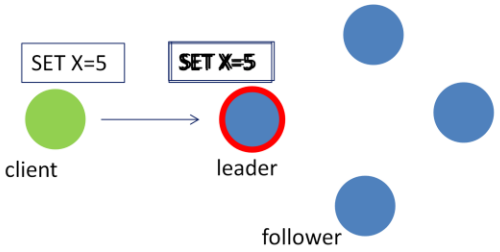
- Consensus algorithms ensures that all logs contain the same commands in the same order
- State machines remain consistent if commands have deterministic results

Consensus algorithms are typically needed in the context of *replicated state machines*. State machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicate state machines are used to solve a variety of fault tolerance problems in distributed systems. For example, large-scale systems that have a single cluster leader, such as HDFS, typically use a separate replicated state machine to manage leader election and store configuration information that must survive crashes. Examples of replicated state machines include Chubby, ZooKeeper and Etcd.

Replicated state machines are typically implemented using a replicated log. Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence. Since the state machines are deterministic, each machine computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

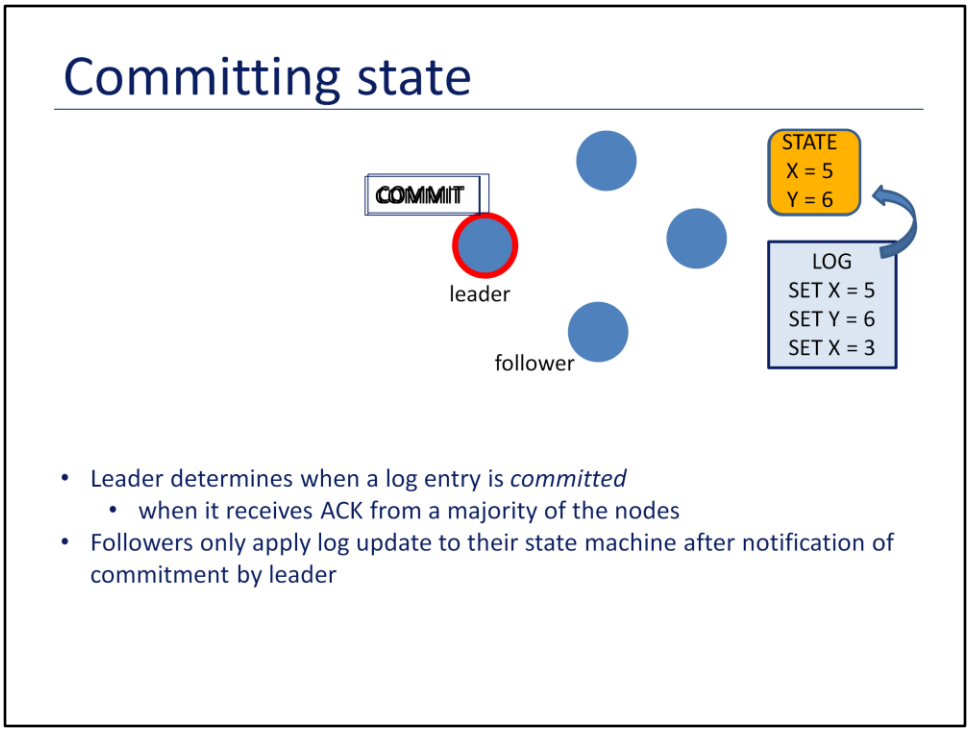
Leaders and followers



- Cluster nodes elect a single **leader**, others become followers
  - Sole server habilitated to accept commands from clients
  - Will enter them in its log and forward them to the followers
  - Tells followers when it's safe to apply log entries to their state machine

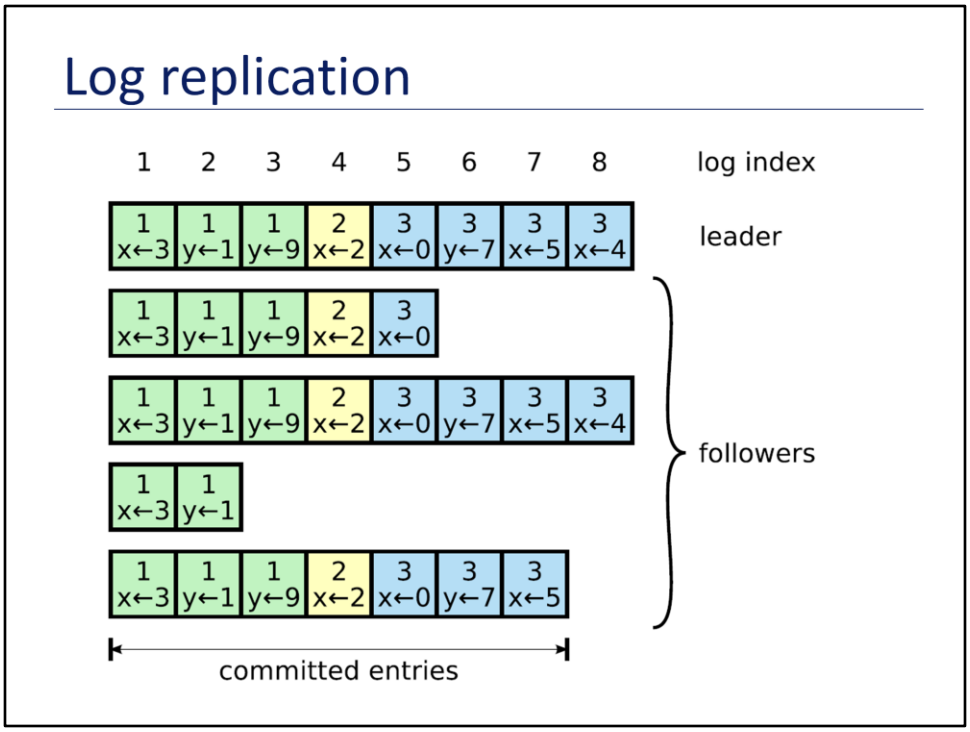
A RAFT cluster consists of several servers (typically five). The cluster can tolerate the failure of any two servers.

Raft implements consensus by first electing one distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows in a simple fashion from the leader to other servers. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

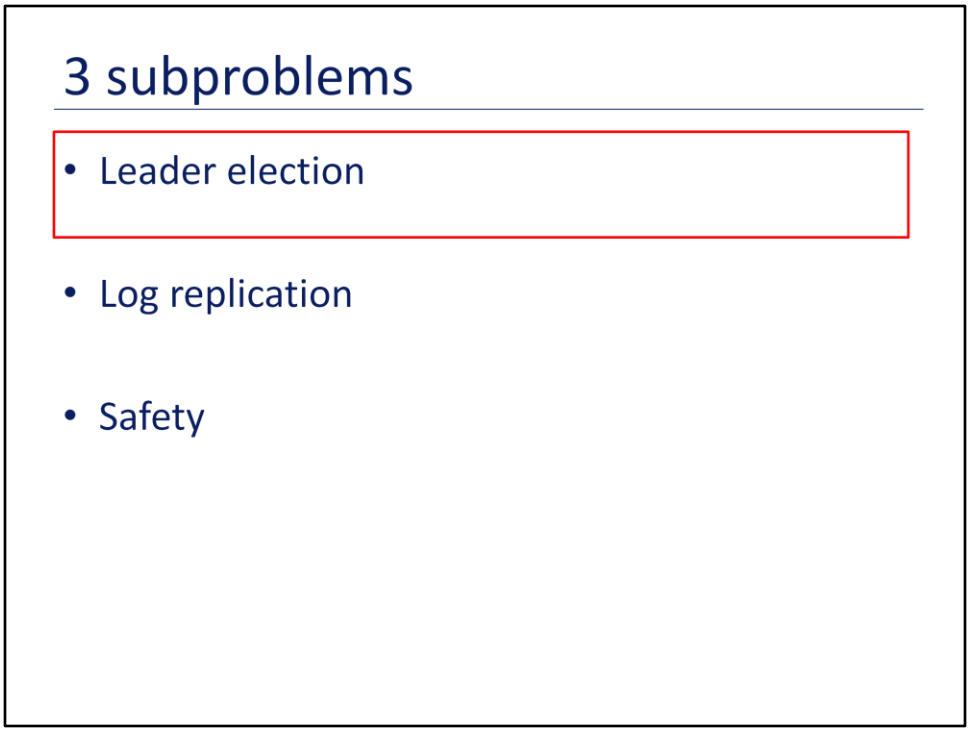


The leader decides when it is safe to apply a log entry to state machines, such an entry is called *committed.* Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers. This also commits all preceding entries in the leader's log, including entries created by previous leaders.

The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in log order).



Each follower has a subset of the committed entries of the leader.

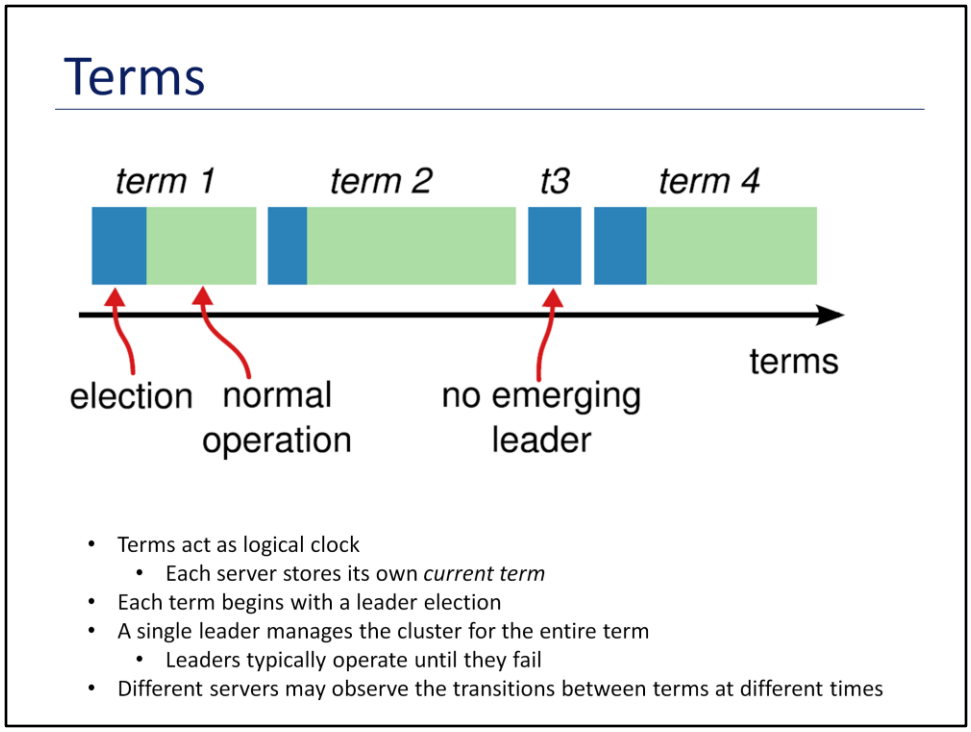


Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

• Leader election: a new leader must be chosen when an existing leader fails

• Log replication: the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own

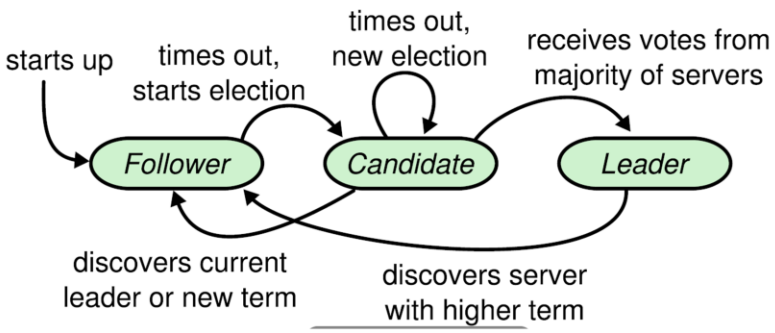
• Safety: if a server has applied a log entry at given index to its state machine, no other server will ever apply a different log entry for the same index



Raft divides time into *terms* of arbitrary length. Terms are numbered with consecutive integers. Each term begins with an *election*, in which one or more candidates attempt to become leader. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term.

Different servers may observe the transitions between terms at different times, and in some situations a server may not observe an election or even entire terms. Terms act as a logical clock in Raft, and they allow servers to detect obsolete information such as stale leaders. Each server stores a *current term* number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

Server state transitions



At any given time each server in a RAFT cluster is in one of three states *leader*, *follower*, or *candidate*. In normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

Leader election: candidates

https://ramcloud.stanford.edu/~ongaro/raftscope/

- After election timeout, follower transitions to candidate
  - increases its term
  - sends "RequestVote" to other cluster members
- Election outcome
  - majority (incl itself) votes for this candidate
    - servers vote on first-come-first-served and vote only once per term
    - candidate becomes leader for this term
  - candidate receives notification of other leader in at least the same term
    - candidate resigns
  - split vote
    - new election procedure
    - randomized time-outs make this very unlikely; but not impossible

Raft uses a heartbeat mechanism to trigger leader election. When servers start up, they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntriesRPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the *election timeout*, then it assumes there is no viable leader and begins an election to choose a new leader.

To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster. A candidate continues in this state until one of three things happens:

(a) it wins the election,

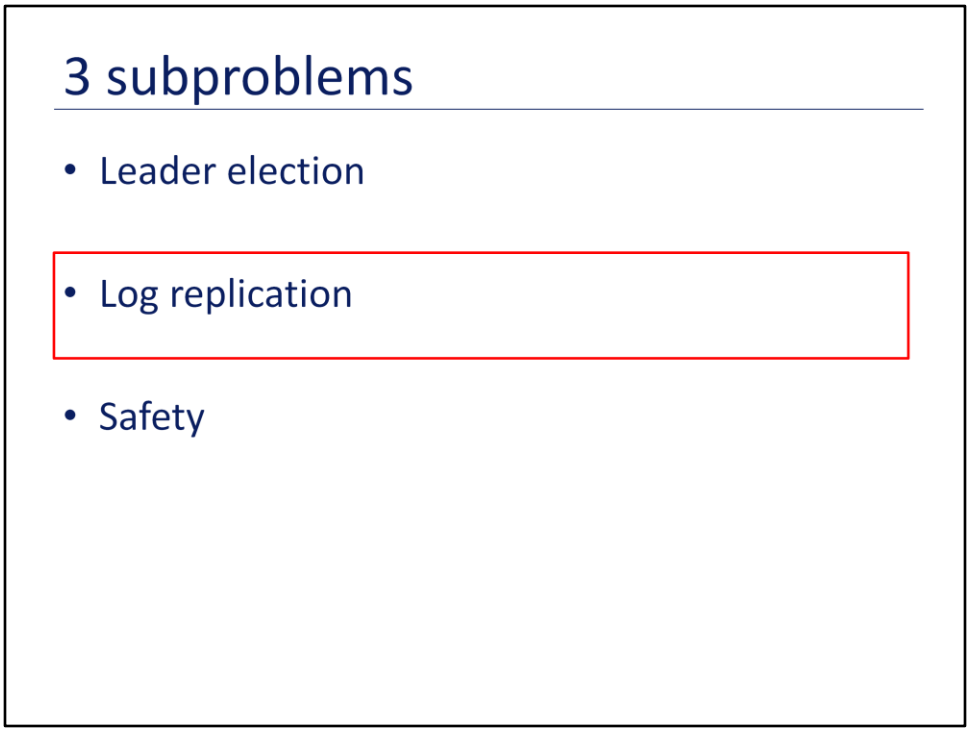
(b) another server establishes itself as leader, or

(c) a period of time goes by with no winner.

A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis. The majority rule ensures that at most one candidate can win the election for a particular term. Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections.

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term (included in its RPC) is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate's current term, then the candidate rejects the RPC and continues in candidate state.

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.
Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election.



Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems:

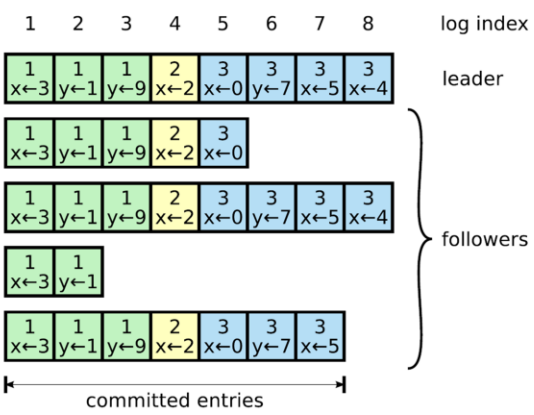
• Leader election: a new leader must be chosen when an existing leader fails

• Log replication: the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own

• Safety: if a server has applied a log entry at given index to its state machine, no other server will ever apply a different log entry for the same index

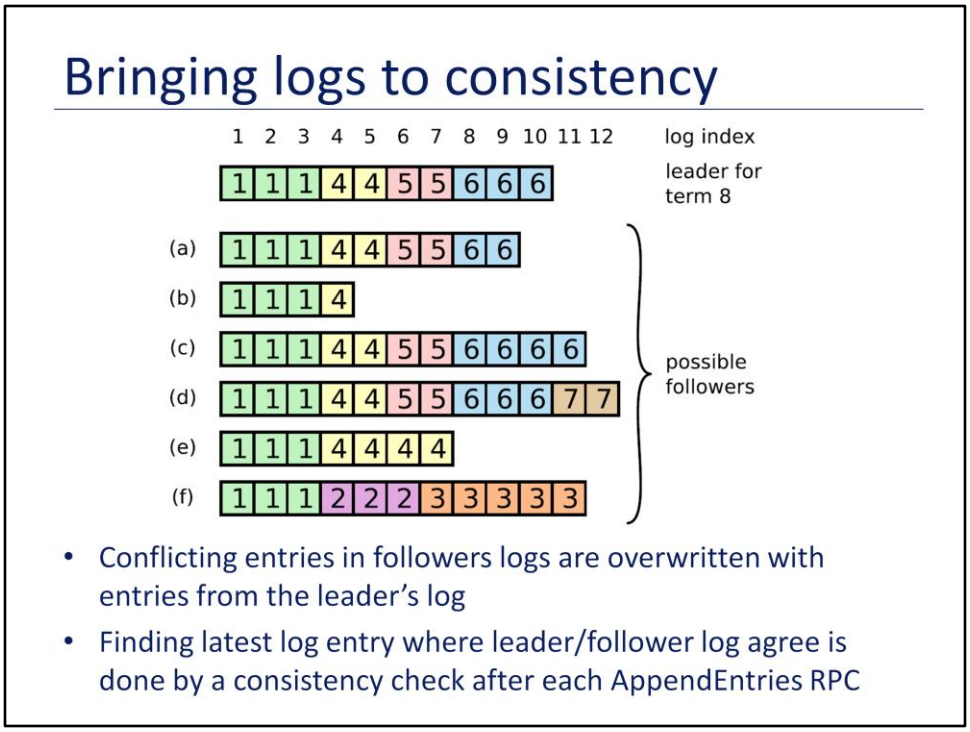
Log replication

- Two entries in different logs having the same index and term store the same command
  - a leader creates at most one entry with a given log index in a given term
- Two entries in different logs having the same index and term are identical in all preceding entries
  - leader includes index/term of previous entry in its log to append request



Once a leader has been elected, it begins servicing client requests. Each client request contains a command to be executed by the replicated state machines. The leader appends the command to its log as a new entry, then issues AppendEntries RPCs in parallel to each of the other servers to replicate the entry. When the entry has been safely replicated, the leader applies the entry to its state machine and returns the result of that execution to the client. If followers crash or run slowly, or if network packets are lost, the leader retries AppendEntries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries.

Logs are organized as shown. Each log entry stores a state machine command along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs. Each log entry also has an integer index identifying its position in the log.



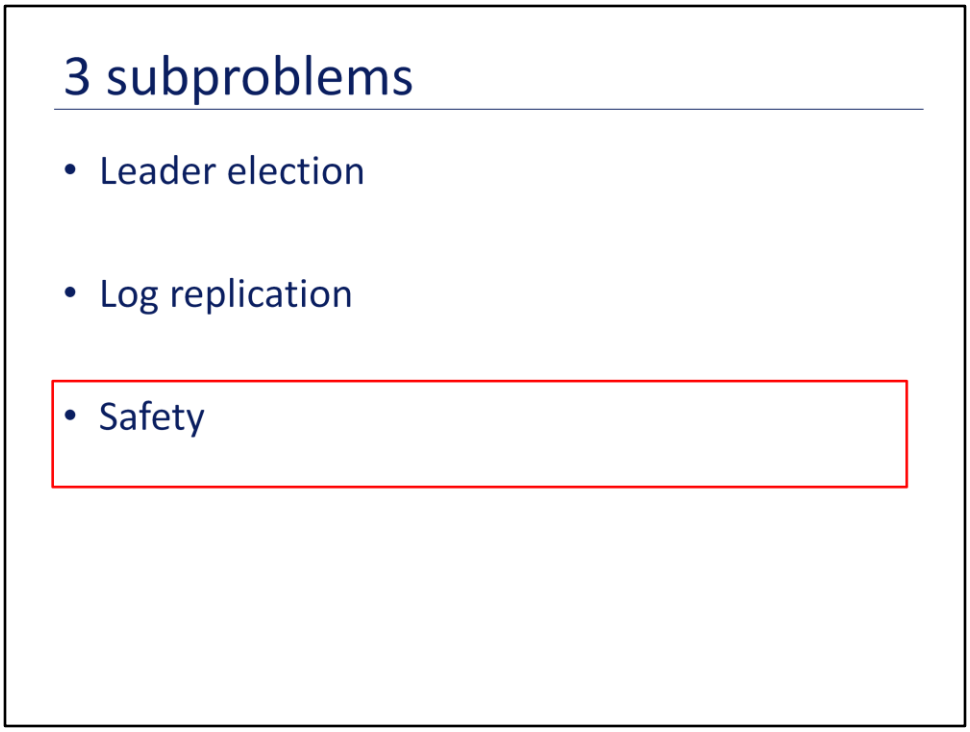
During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails. However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes. The above figure illustrates the ways in which followers'logs may differ from that of a new leader. A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

When the leader at the top in the figure comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries

in the follower's log after that point, and send the follower all of the leader's entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs. The leader maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log (11 in the figure above). If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC. After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log (if any). Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.

With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consistency check. A leader never overwrites or deletes entries in its own log.



The mechanisms described so far are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences.

Election restriction

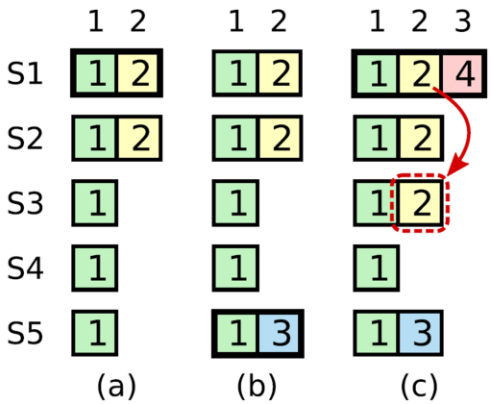
- Candidate cannot win election unless its log contains all committed entries
  - avoids transfer of entries between old and new leader
- Intuitive proof:
  - candidate must contact a majority of the cluster to be elected
  - every committed update is present in at least one of those servers
  - voter denies its vote if own log is more up-to-date

Raft uses a simple approach that guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs.

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries. A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log in that majority (where "up-to-date" is defined precisely below), then it will hold all the committed entries. The RequestVote RPC implements this restriction: the RPC includes information about the candidate's log, and the voter denies its vote if its own log is more up-to-date than that of the candidate. Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

Committing entries from previous terms

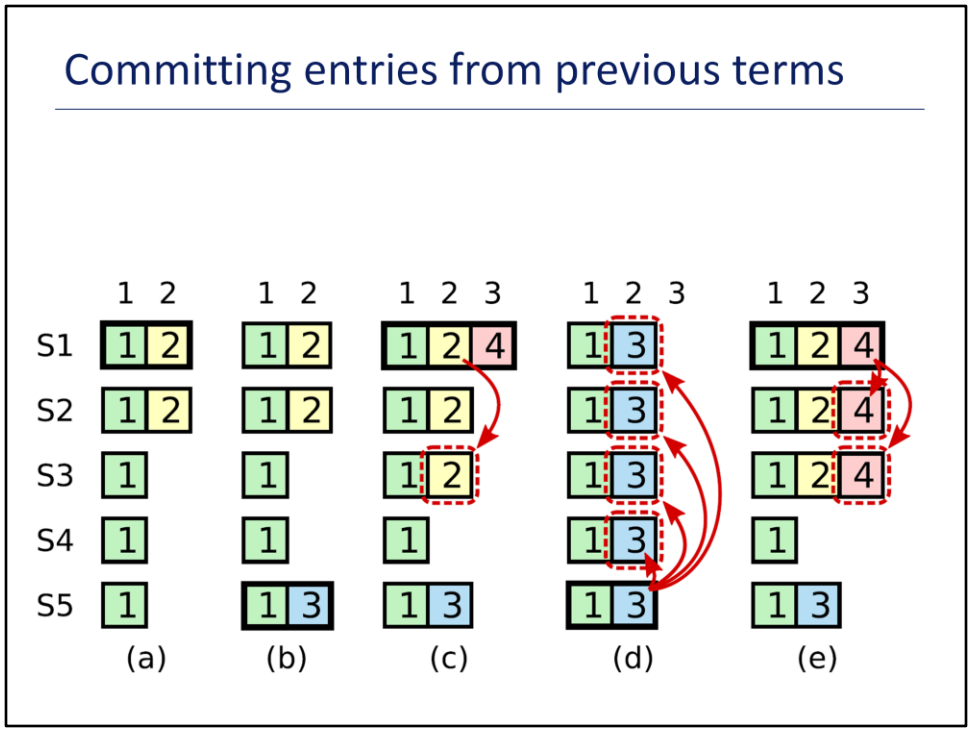
- A leader knows that an entry from its current term is committed once stored on a majority of servers
- A leader cannot determine commitment of entries from previous terms based on this majority rule



Log entry from term 2 has been replicated on a majority of servers, but is not replicated.

A leader knows that an entry from its current term is committed once that entry is stored on a majority of the servers. If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry. However, a leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers.

To illustrate this problem, we use the above time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. It can be overwritten by a future leader (as we show on the next slide)



If S1 crashes, as in (d), S5 could be elected leader (with votes from S2, S3 and S4) and overwrite the entry with its own entry from term 3.
However, if S1 replicates an entry from its **current** term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

Thus, to eliminate commitment problems with previous terms, Raft never commits log entries from previous terms by counting replicas. Only log entries from the leader's current term are committed by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly (because servers will first synchronize their complete log with the leader, as described before).

Bibliography: distributed systems

- D. Thain, Foundations of Distributed Systems
- The paper trail, A brief tour of FLP impossibility, http://the-paper-trail.org/blog/a-brief-tour-of-flp-impossibility/
- A. Brown, Distributed agreement, http://www.cs.toronto.edu/~demke/469F.07/Lectures/Lecture18.ppt
- R. Wattenhofer, Principles of Distributed Computing (lecture collection), http://disco.ethz.ch/lectures/podc_allstars/