UCL
Université catholique de Louvain

INGI2145: CLOUD COMPUTING (Fall 2014)

Design for scale

2 October 2014

Certain lecture slides adapted from Upenn NETS212 by A. Haeberlen, Z. Ives
Reproduced with permission

Announcements

- VMs in the computer rooms
  - Please send me a note (via Piazza) if you want to run the VM in the INGI computers

Plan for today

- Parallel programming and its challenges NEXT
  - Parallelization and scalability, Amdahl's law
  - Synchronization, consistency
  - Architectures: SMP, NUMA, Shared-Nothing
- Wide-area network
  - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency

Goal: Building fast, scalable systems

- What are the hard problems?
- How do they affect what we can do?
- How do we write the software? How do we speed it up?

What is scalability?

- A system is scalable if it can easily adapt to increased (or reduced) demand
  - Example: A storage system might start with a capacity of just 10TB but can grow to many PB by adding more nodes
  - Scalability is usually limited by some sort of bottleneck
- Often, scalability also means...
  - the ability to operate at a very large scale
  - the ability to grow efficiently
    - Example: 4x as many nodes → ~4x capacity (not just 2x!)

An obvious way to scale

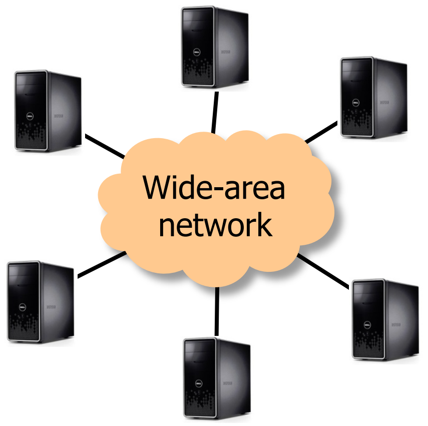
- Throw more hardware at the problem!
- go from one processor to many
- go from one computer to many
- … but what are the challenges?

Scale increases complexity

Single-core machine

Multi-core server

Cluster



Large-scale distributed system

More challenges

Time-sharing

True concurrency

Network
Message passing
More failure modes (faulty nodes, ...)

Wide-area network
Even more failure modes
Incentives, laws, ...

A big challenge of scale

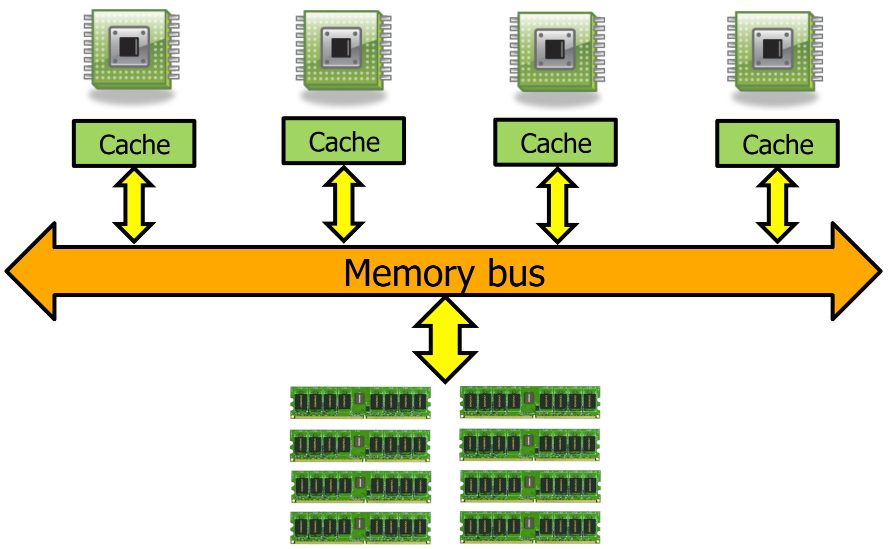
"when a system processes trillions and trillions of requests, events that normally have a low probability of occurrence are now guaranteed to happen and need to be accounted for up front in the design and architecture of the system"

– Werner Wogels, Amazon.com

Example: $P(\text{any failure}) = 1 - P(\text{individual node not failing})^{\text{number of nodes}}$

- Single node 99.9% chance of not failing, a cluster of 40 has 96.07% chance of not failing
- 4% chance that something will fail!

Parallelism in one machine with Symmetric Multiprocessing (SMP)



- For now, assume we have multiple cores that can access the same shared memory
  - Any core can access any byte; speed is uniform (no byte takes longer to read or write than any other)
  - Not all machines are like that -- other models discussed later

Parallelization

```
void bubblesort(int nums[]) {
  boolean done = false;
  while (!done) {
    done = true;
    for (int i=1; i<nums.length; i++) {
      if (nums[i-1] > nums[i]) {
        swap(nums[i-1], nums[i]);
        done = false;
      }
    }
  }
}
```

```
int[] mergesort(int nums[]) {
  int numPieces = 10;
  int pieces[][] = split(nums, numPieces);
  for (int i=0; i<numPieces; i++)
    sort(pieces[i]);
  return merge(pieces);
}
```

Can be done in parallel!

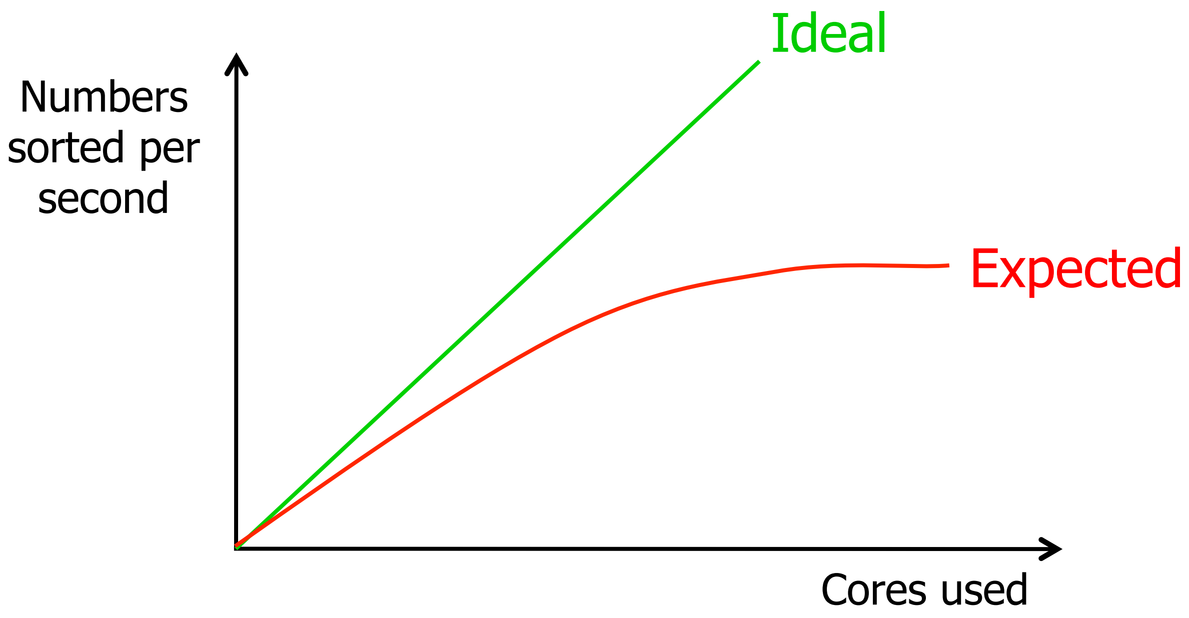
- The left algorithm works fine on one core
- Can we make it faster on multiple cores?
  - Difficult – need to find something for the other cores to do
  - There are other sorting algorithms where this is much easier
  - Not all algorithms are equally parallelizable
  - Can you have scalability without parallelism?

Scalability

Speedup: $S_N = \frac{T_1}{T_n}$

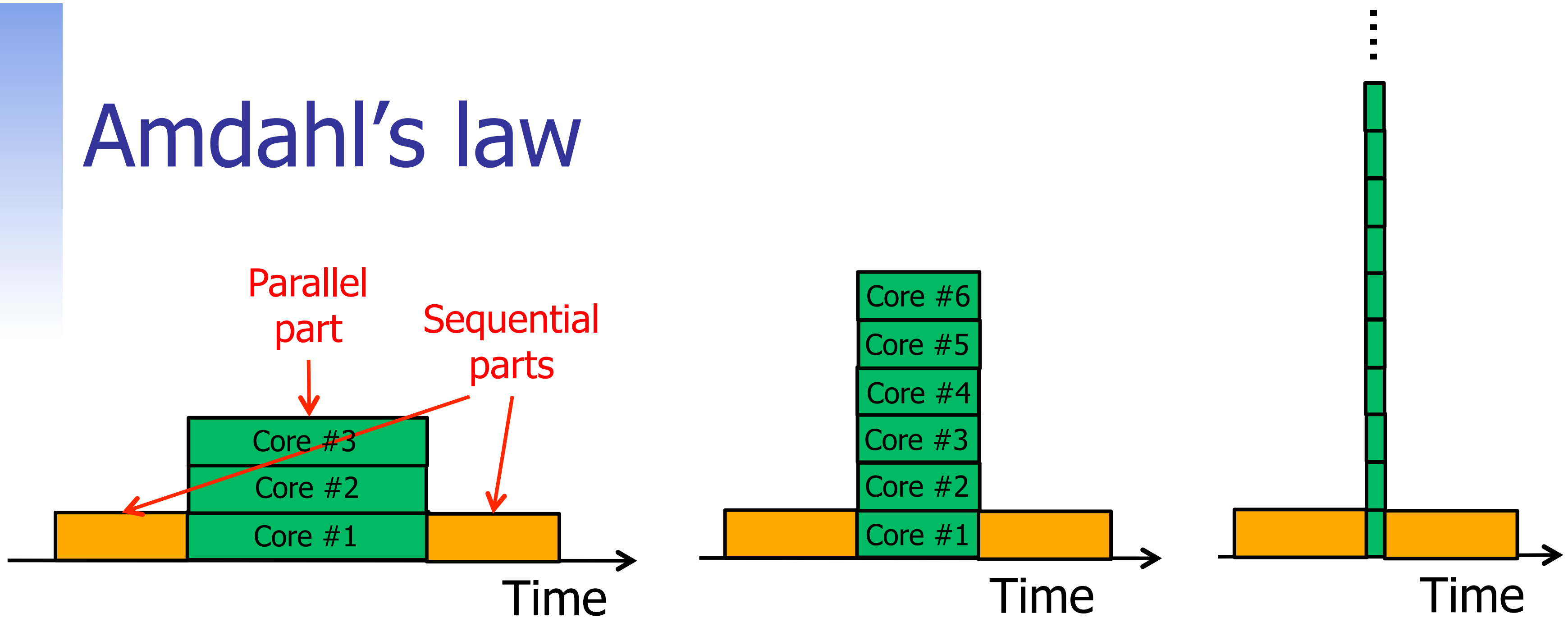
Completion time with one core

Completion time with n cores



- If we increase the number of processors, will the speed also increase?
  - Yes, but (in almost all cases) only up to a point
  - Why?

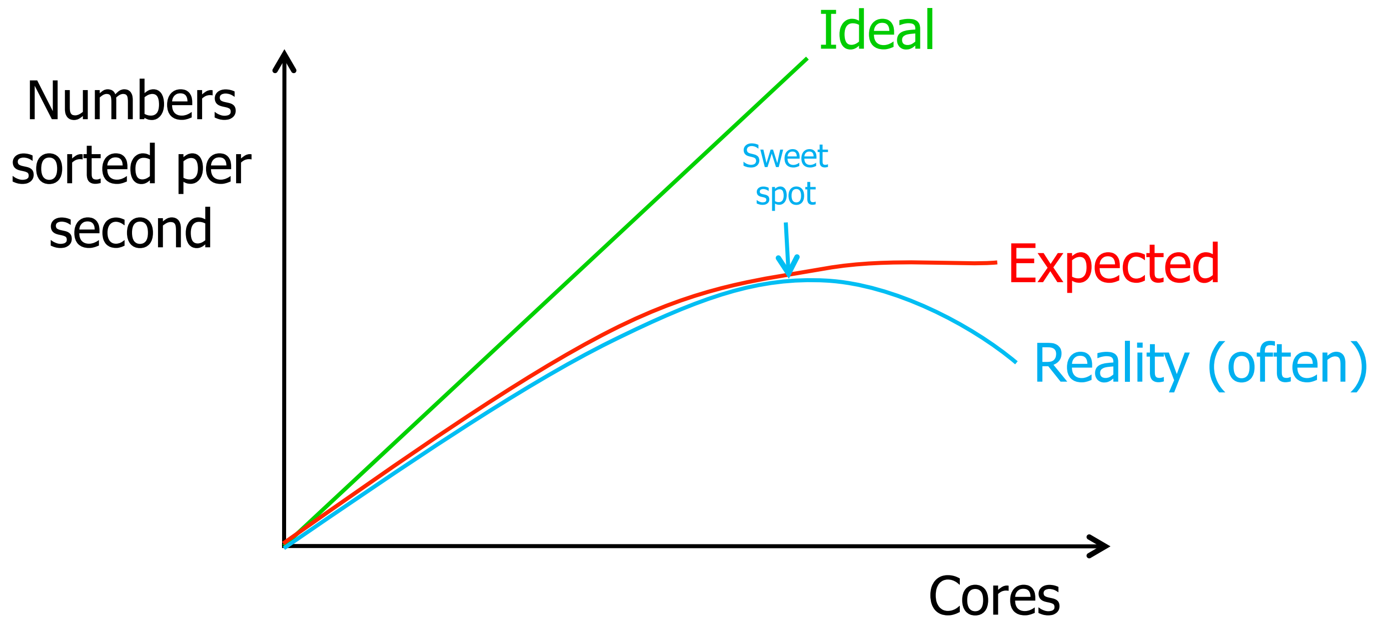
Amdahl's law



- Usually, not all parts of the algorithm can be parallelized
- Let α be the fraction of the algorithm that is sequential, then the maximum speedup *S* that can be achieved with N cores is:

$$S = \frac{1}{\alpha + \frac{1-\alpha}{N}}$$

Is more parallelism always better?

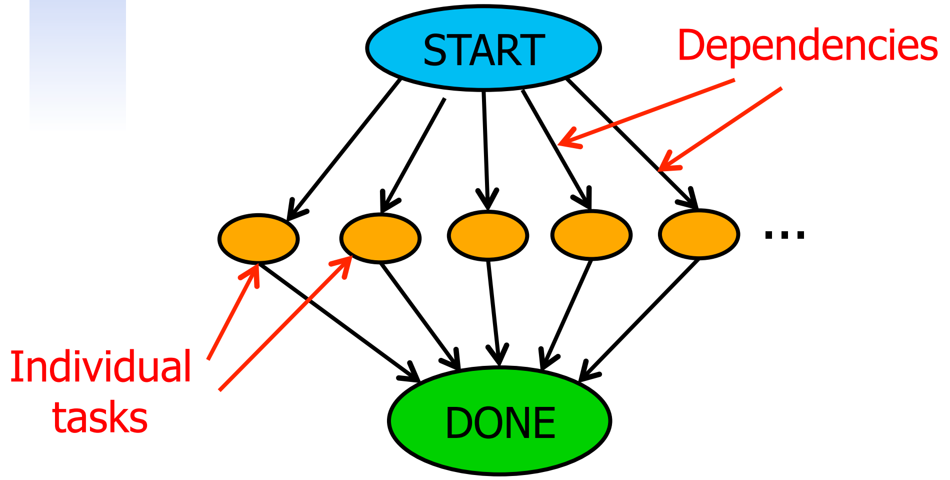


- Increasing parallelism beyond a certain point can cause performance to decrease! Why?
- Time for serial parts can depend on #cores
  - Example: Need to send a message to each core to tell it what to do

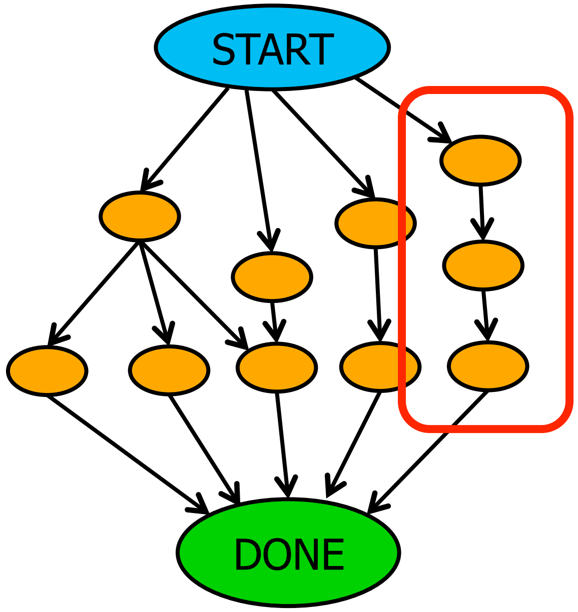
Granularity

- How big a task should we assign to each core?
  - Coarse-grain vs. fine-grain parallelism
- Frequent coordination creates overhead
  - Need to send messages back and forth, wait for other cores...
  - Result: Cores spend most of their time communicating
- Coarse-grain parallelism is usually more efficient
  - Bad: Ask each core to sort three numbers
  - Good: Ask each core to sort a million numbers

Dependencies



"Embarrassingly parallel"



With dependencies

- What if tasks depend on other tasks?
  - Example: Need to sort lists <u>before</u> merging them
  - Limits the degree of parallelism
  - Minimum completion time (and thus maximum speedup) is determined by the longest path from start to finish
    - Assumes resources are plentiful; actual speedup may be lower

Heterogeneity

- What if...
  - some tasks are larger than others?
  - some tasks are harder than others?
  - some tasks are more urgent than others?
  - not all cores are equally fast, or have different resources?

- Result: Scheduling problem
  - Can be very difficult

Recap: Parallelization

- Parallelization is hard
  - Not all algorithms are equally parallelizable – need to pick very carefully

- Scalability is limited by many things
  - Amdahl's law
  - Communication and coordination overheads
  - Dependencies between tasks
  - Heterogeneity
  - ...

Plan for today

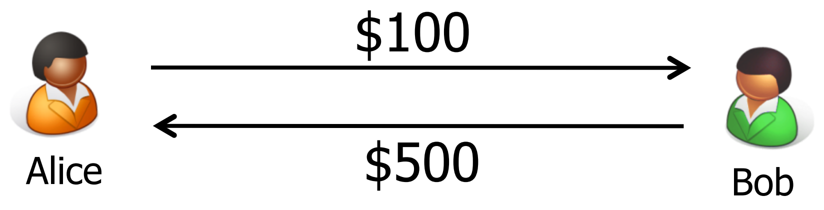
- Parallel programming and its challenges
  - Parallelization and scalability, Amdahl's law ✓
  - Synchronization, consistency ← NEXT
  - Architectures: SMP, NUMA, Shared-Nothing
- Wide-area network
  - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency

Why do we need synchronization?

```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

- Simple example: Accounting system in a bank
  - Maintains the current balance of each customer’s account
  - Customers can transfer money to other customers

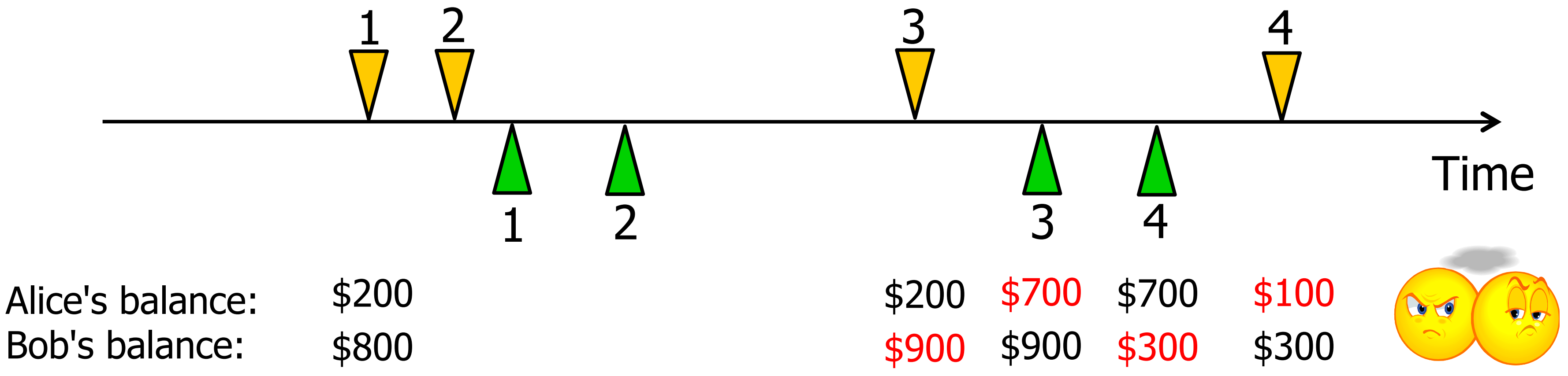
Why do we need synchronization?



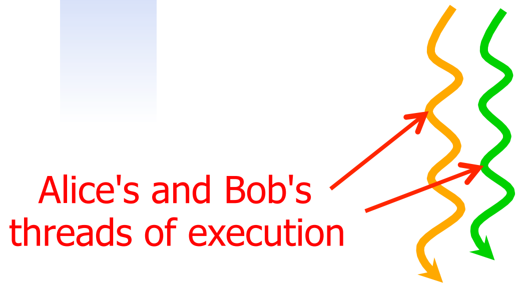
```
1)  B=Balance(Bob)
2)  A=Balance(Alice)
3)  SetBalance(Bob,B+100)
4)  SetBalance(Alice,A-100)
```

```
1)  A=Balance(Alice)
2)  B=Balance(Bob)
3)  SetBalance(Alice,A+500)
4)  SetBalance(Bob,B-500)
```

- What can happen if this code runs concurrently?



Problem: Race condition



```
void transferMoney(customer A, customer B, int amount)
{
  showMessage("Transferring "+amount+" to "+B);
  int balanceA = getBalance(A);
  int balanceB = getBalance(B);
  setBalance(B, balanceB + amount);
  setBalance(A, balanceA - amount);
  showMessage("Your new balance: "+(balanceA-amount));
}
```

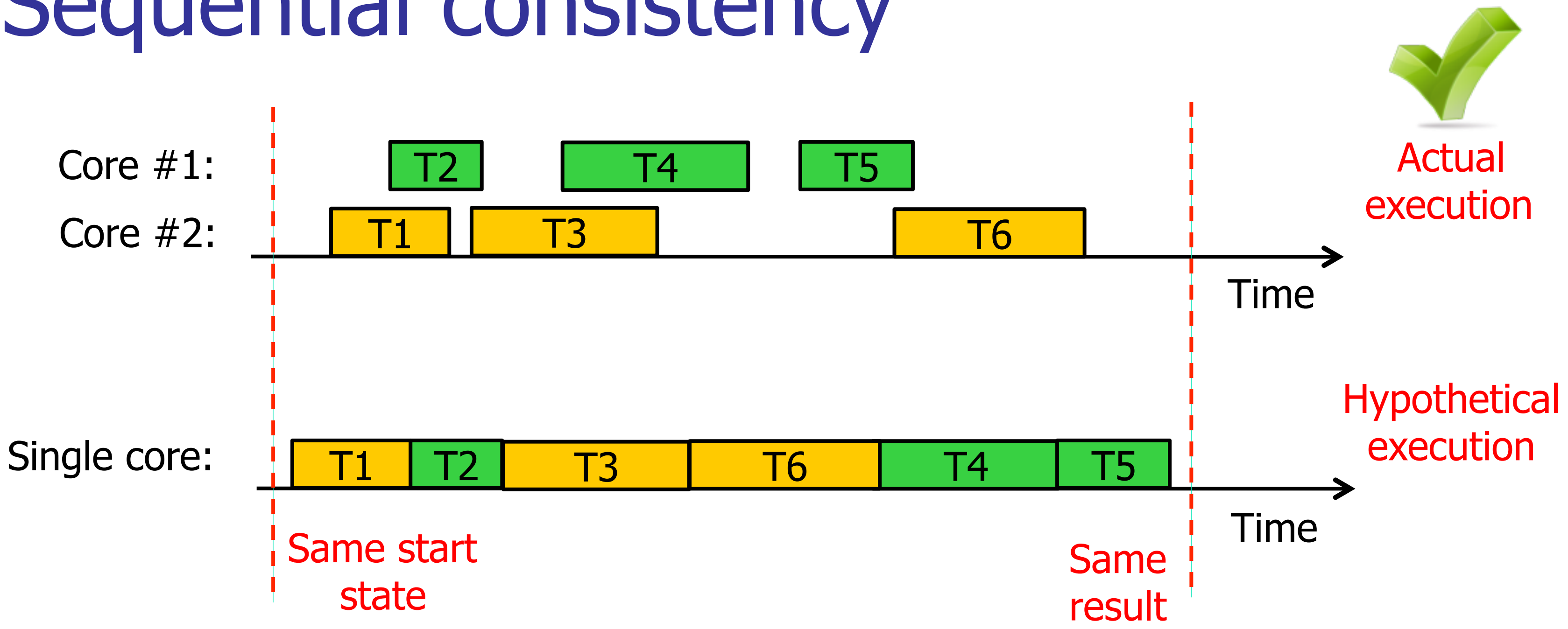
- What happened?
  - Race condition: Result of the computation depends on the exact timing of the two threads of execution, i.e., the order in which the instructions are executed
  - Reason: Concurrent <u>updates</u> to the same state
    - Can you get a race condition when all the threads are reading the data, and none of them are updating it?

Goal: Consistency

- What should have happened?
  - Intuition: It shouldn't make a difference whether the requests are executed concurrently or not

- How can we formalize this?
  - Need a consistency model that specifies how the system should behave in the presence of concurrency

Sequential consistency



- Sequential consistency:
  - The result of any execution is the same as if the operations of all the cores had been executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by the program

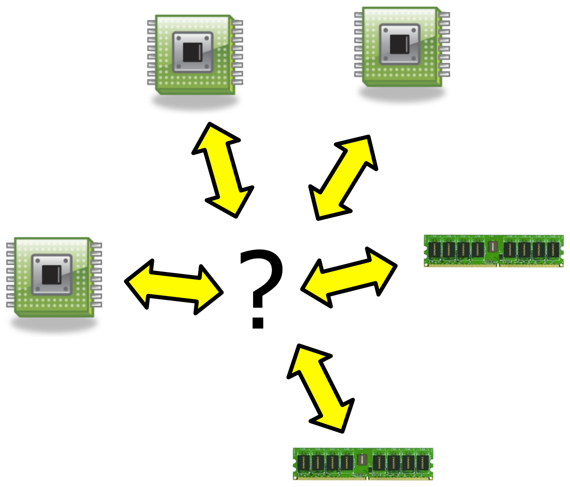
Other consistency models

- Strong consistency
  - After update completes, all subsequent accesses will return the updated value
- Weak consistency
  - After update completes, accesses do not necessarily return the updated value; some condition must be safisfied first
    - Example: Update needs to reach all the replicas of the object
- Eventual consistency
  - Specific form of weak consistency: If no more updates are made to an object, then eventually all reads will return the latest value
    - Variants: Causal consistency, read-your-writes, monotonic writes, ...
- How do we build systems that achieve this?

Plan for today

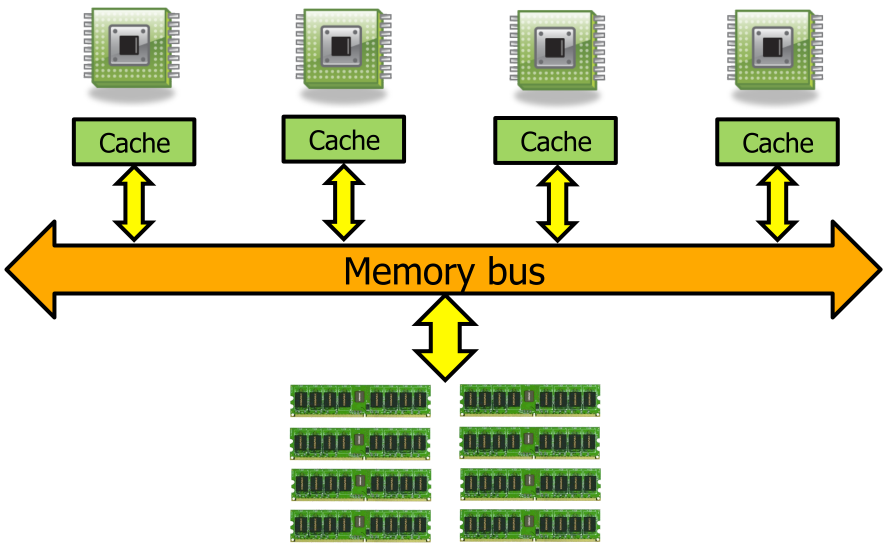
- Parallel programming and its challenges
  - Parallelization and scalability, Amdahl's law ✓
  - Synchronization, consistency ✓
  - Architectures: SMP, NUMA, Shared-Nothing NEXT
- Wide-area network
  - Latency, packet loss, bottlenecks, and why they matter
- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency

Other architectures



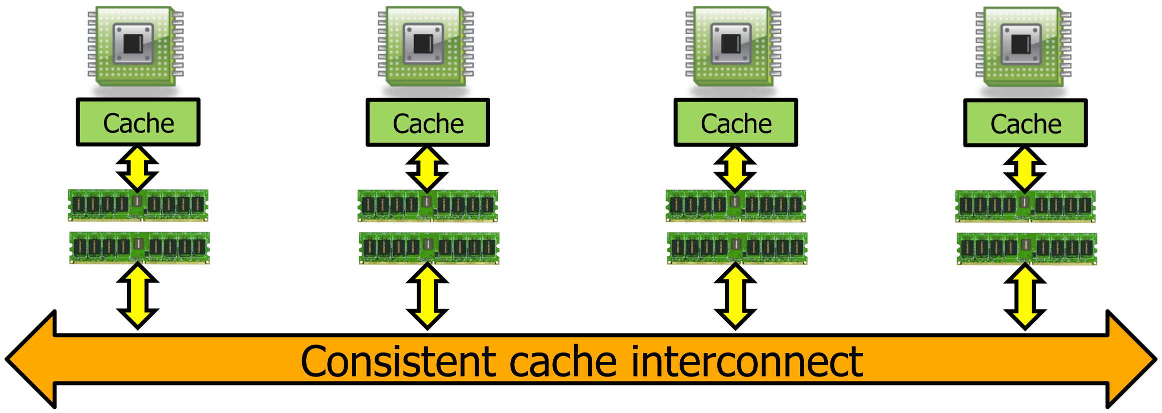
- Earlier assumptions:
  - All cores can access the same memory
  - Access latencies are uniform
- Why is this problematic?
  - Processor speeds in GHz, speed of light is 299 792 458 m/s
    → Processor can do 1 addition while signal travels 30cm
    → Putting memory very far away is not a good idea
- Let's talk about other ways to organize cores and memory!
  - SMP, NUMA, Shared-nothing

Symmetric Multiprocessing (SMP)



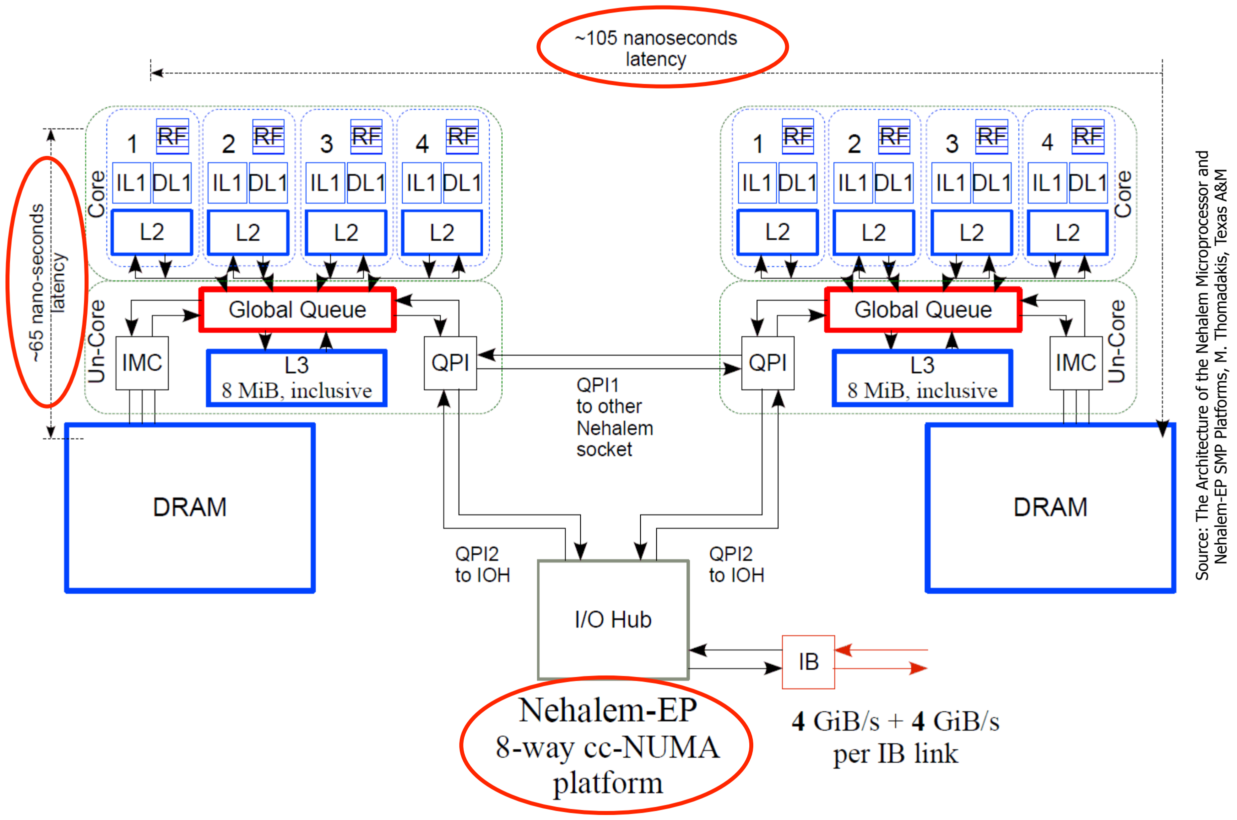
- All processors share the same memory
  - Any CPU can access any byte; latency is always the same
  - Pros: Simplicity, easy load balancing
  - Cons: Limited scalability (~12 processors), expensive

Non-Uniform Memory Architecture (NUMA)



- Memory is local to a specific processor
  - Each CPU can still access any byte, but accesses to 'local' memory are considerably faster (2-3x)
  - Pros: Better scalability
  - Cons: Complicates programming a bit, scalability still limited

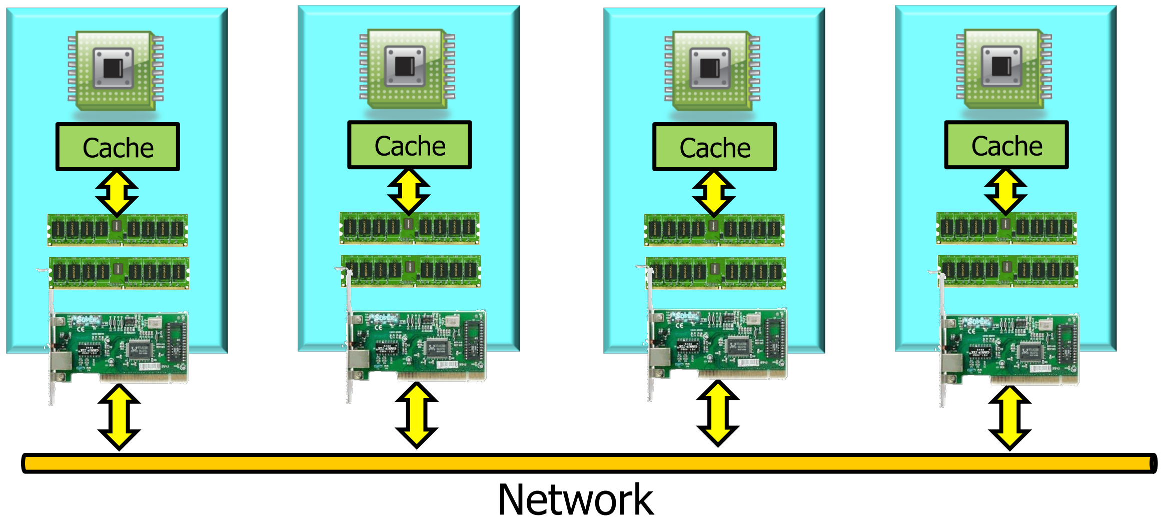
Example: Intel Nehalem



~105 nanoseconds latency
~65 nano-seconds latency
Core
1 RF IL1 DL1 L2
2 RF IL1 DL1 L2
3 RF IL1 DL1 L2
4 RF IL1 DL1 L2
Un-Core
Global Queue
IMC
L3 8 MiB, inclusive
QPI
QPI1 to other Nehalem socket
DRAM
QPI2 to IOH
I/O Hub
IB
Nehalem-EP 8-way cc-NUMA platform
4 GiB/s + 4 GiB/s per IB link
Source: The Architecture of the Nehalem Microprocessor and Nehalem-EP SMP Platforms, M. Thomadakis, Texas A&M

- Access to remote memory is slower
  - In this case, 105ns vs 65ns

Shared-Nothing

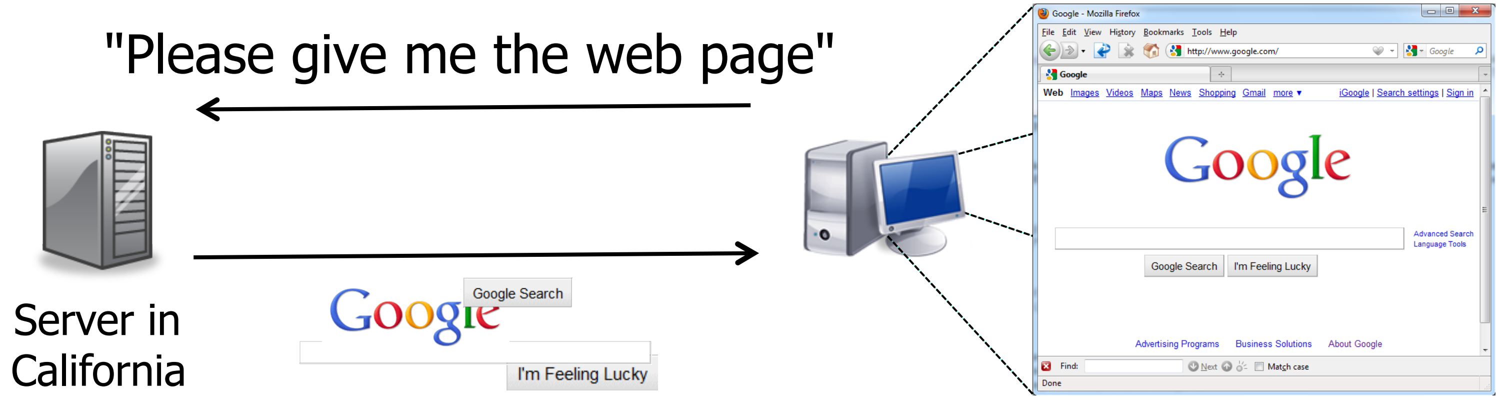


- Independent machines connected by network
  - Each CPU can only access its local memory; if it needs data from a remote machine, it must send a message there
  - Pros: Much better scalability
  - Cons: Requires a different programming model

Plan for the next two lectures

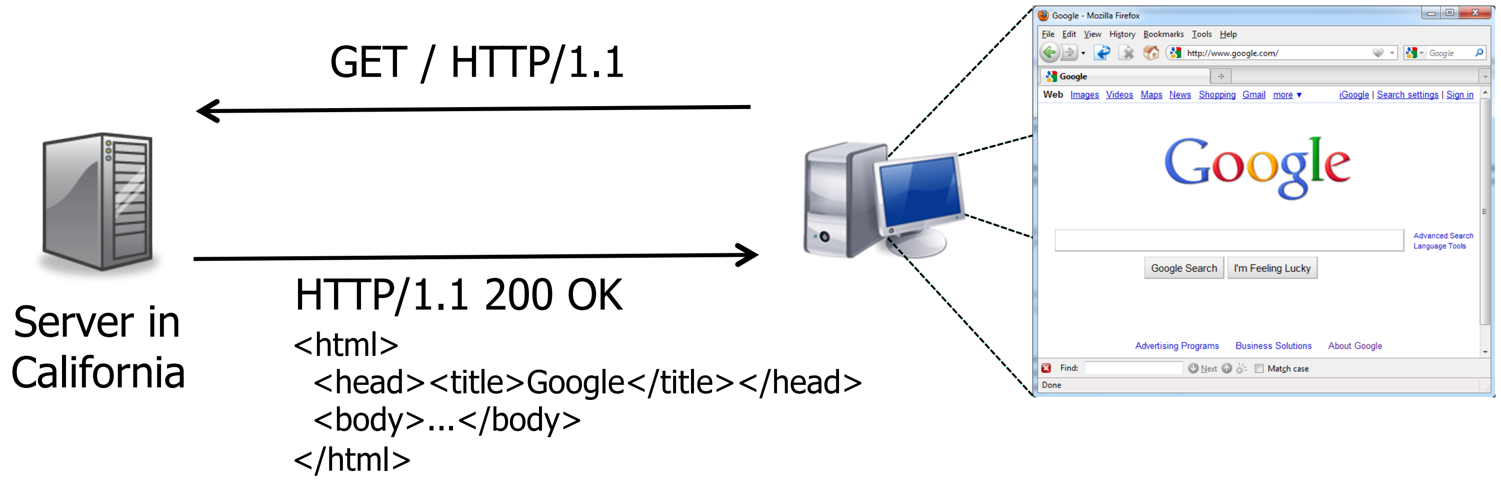
- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Synchronization, consistency ✓
  - Architectures: SMP, NUMA, Shared-nothing ✓
- Wide-area network
  - Latency, packet loss, bottlenecks, and why they matter ← NEXT
- Distributed programming and its challenges
  - Faults, failures, and what we can do about them
  - Network partitions, CAP theorem, relaxed consistency

The life and times of a web request



- What happens when I open the web page 'www.google.com' in my browser?
  - First approximation: My computer contacts another computer in California and requests the web page from there

HTTP and HTML

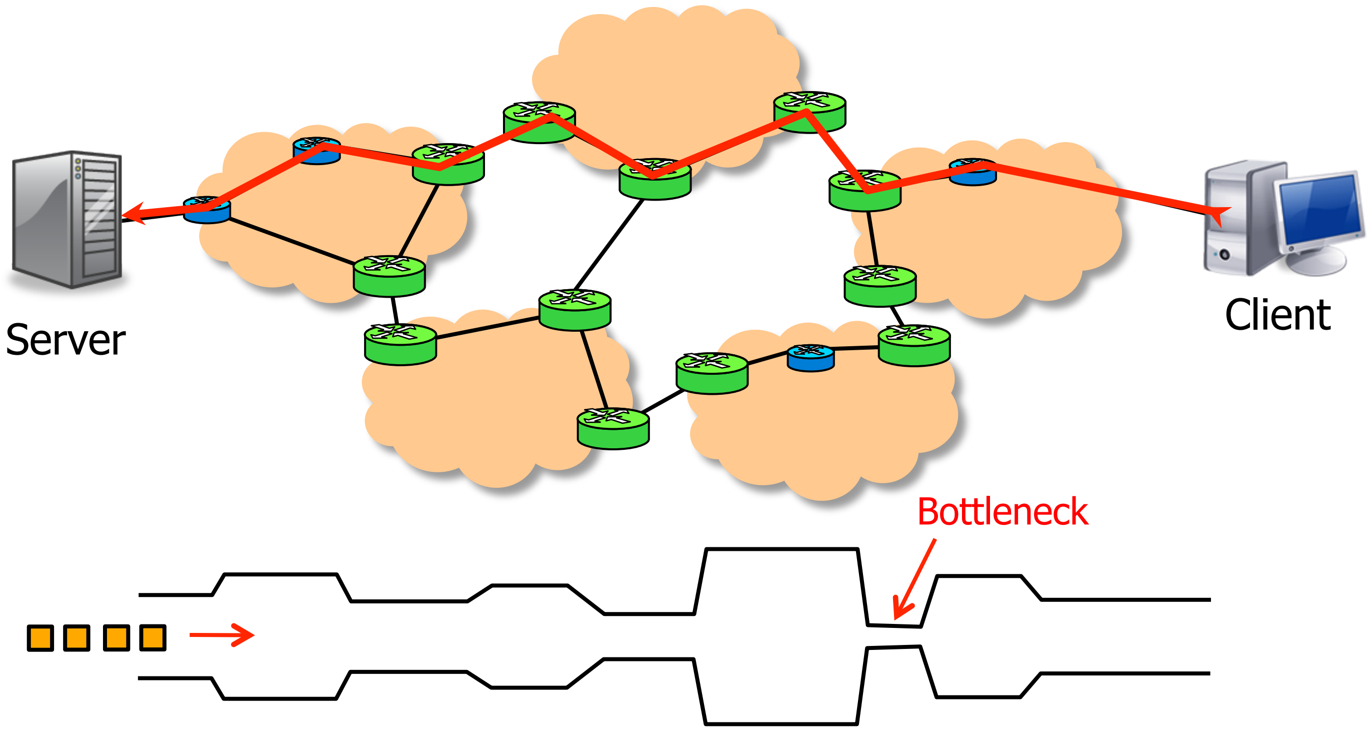


```
GET / HTTP/1.1
```

```
HTTP/1.1 200 OK
<html>
  <head><title>Google</title></head>
  <body>...</body>
</html>
```

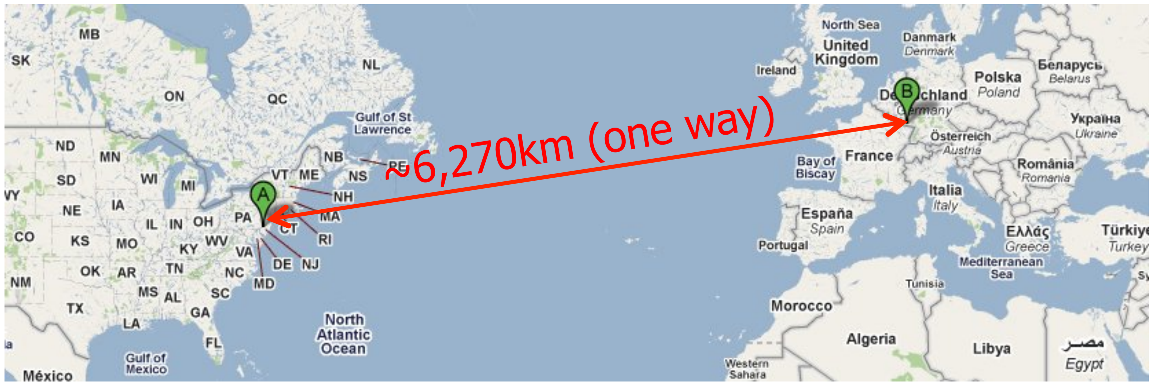
- There are standardized protocols:
  - Hypertext Transfer Protocol (HTTP): Describes how web pages are requested
  - Hypertext Markup Language (HTML): The language the actual web page is written in
- How does the request make it to California?

Path properties: Bottleneck capacity



- How fast can we send data on our path?
  - Limited by the bottleneck capacity
  - What else does the available capacity depend on?
  - Which links are usually the bottleneck links?

Path properties: Propagation delay



```
[ahae@ds01 ~]$ traceroute www.mpi-sws.org
traceroute to www.mpi-sws.org (139.19.1.156), 30 hops max, 60 byte packets
 1  SUBNET-46-ROUTER.seas.UPENN.EDU (158.130.46.1)  1.744 ms  2.134 ms  2.487 ms
 2  158.130.21.34 (158.130.21.34)  5.327 ms  5.395 ms  5.649 ms
 3  isc-uplink-2.seas.upenn.edu (158.130.128.2)  5.671 ms  5.825 ms  6.175 ms
 4  external3-core1.dccs.UPENN.EDU (128.91.9.2)  6.007 ms  6.283 ms  6.362 ms
 5  external-core2.dccs.upenn.edu (128.91.10.1)  6.830 ms  6.990 ms  7.080 ms
 6  local.upenn.magpi.net (216.27.100.73)  7.250 ms  3.429 ms  3.533 ms
 7  remote.internet2.magpi.net (216.27.100.54)  4.487 ms  3.002 ms  2.925 ms
 8  198.32.11.51 (198.32.11.51)  90.557 ms  90.806 ms  91.028 ms
 9  so-6-2-0.rt1.fra.de.geant2.net (62.40.112.57)  97.403 ms  97.473 ms  97.766 ms
10  dfn-gw.rt1.fra.de.geant2.net (62.40.124.34)  98.834 ms  98.890 ms  99.043 ms
11  xr-fzk1-te2-3.x-win.dfn.de (188.1.145.50)  100.627 ms  101.034 ms  101.387 ms
12  xr-kai1-te1-1.x-win.dfn.de (188.1.145.102)  103.985 ms  104.383 ms  104.528 ms
13  xr-saa1-te1-1.x-win.dfn.de (188.1.145.97)  103.636 ms  103.903 ms  104.139 ms
14  kr-0unisb.x-win.dfn.de (188.1.234.38)  103.983 ms  103.746 ms  103.853 ms
15  mpi2rz-hsrp2.net.uni-saarland.de (134.96.6.28)  104.469 ms  104.355 ms  104.491 ms
[ahae@ds01 ~]$
```

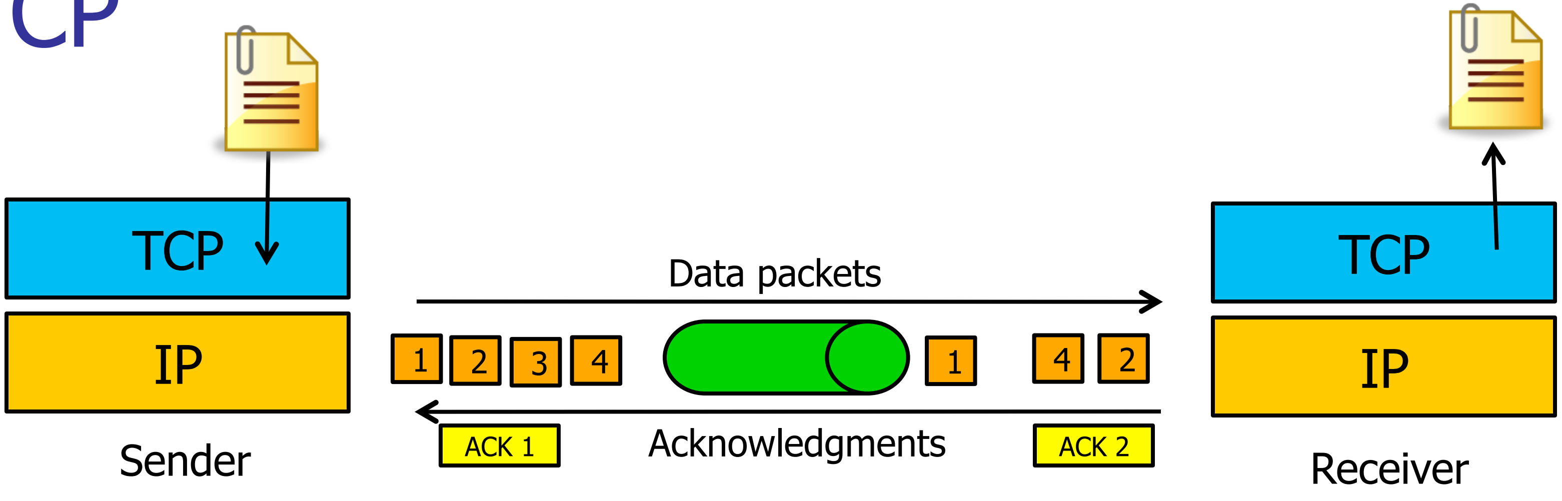
Round-trip time

- Speed of light: 299 792 458 m/s
  - Latency matters!

Path properties: Queueing delay, loss

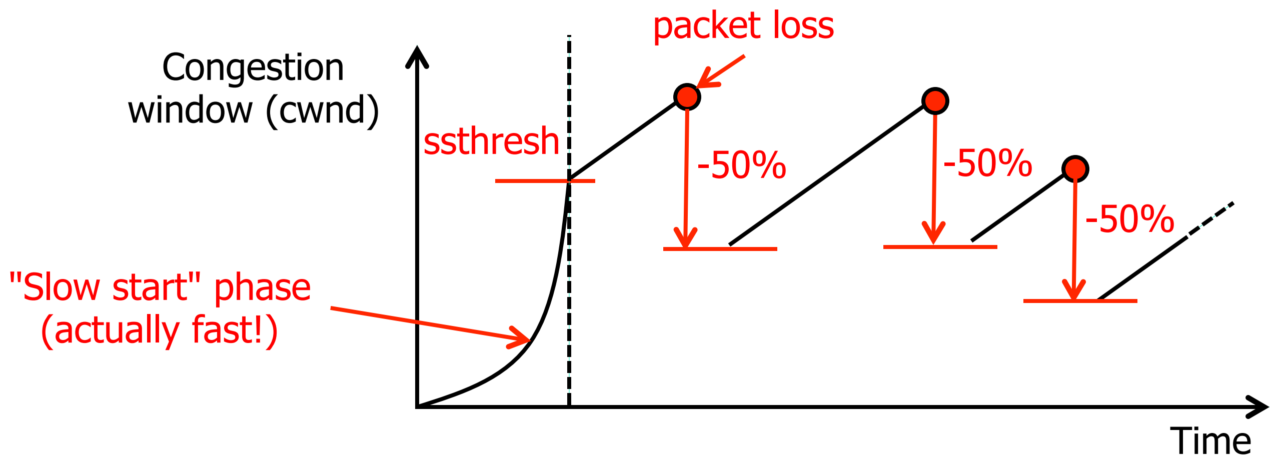
- What if we send packets too quickly?
  - Router stores the packets in a queue until it can send them
  - Consequence : End-to-end delay increases
  - Where does this matter?
- What if the router runs out of queue space?
  - Packets are dropped and lost
- Other reasons why packets might be dropped?

TCP



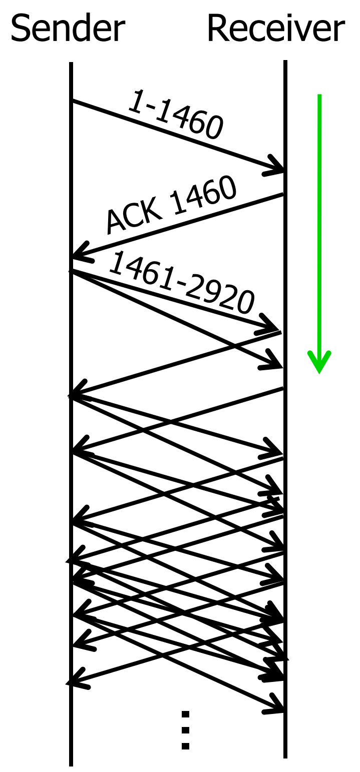
- Transmission Control Protocol (TCP) provides abstraction of a reliable stream of bytes
  - Ensures packets are delivered to application in correct order
  - Retransmits lost packets
  - Tracks available capacity and prevents packets from being sent too fast (congestion control)
  - Prevents sender from overwhelming the receiver (flow control)

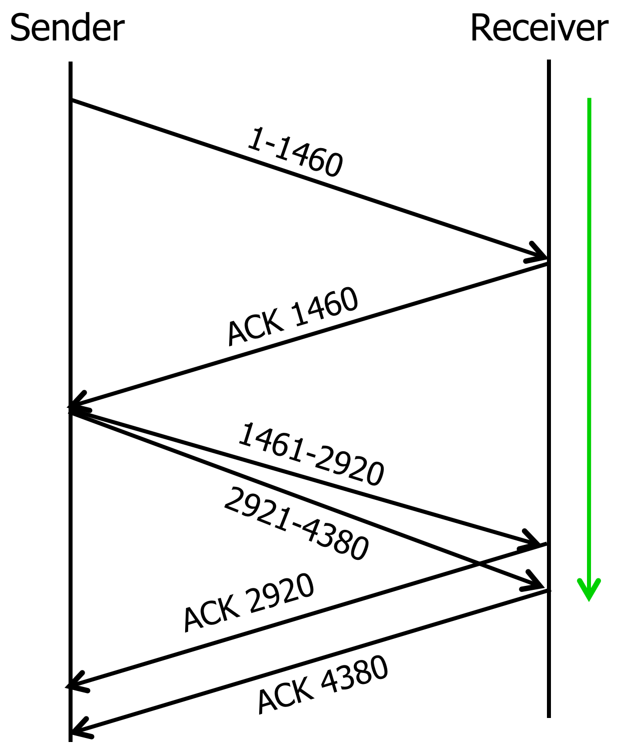
TCP congestion control



- How fast should the sender send?
  - Problem: Available capacity not known (and can vary)
- Solution: Congestion control
  - Maintain a congestion window of max #packets in flight
  - Slow start: Exponential increase until threshold
    - Increase cwnd by one packet for each incoming ACK
  - Congestion avoidance: Additive increase, multiplicative decrease (AIMD)

Another reason why latency matters





- The higher the RTT, the slower the process

Recap: Wide-area network

- How does the network matter to applications?
  - Propagation delay → Good to be physically close to customer
  - Bottlenecks → Transfer speed is limited
  - Queueing delays, loss, reordering → Delay can vary
  - Some of these can be taken care of by TCP

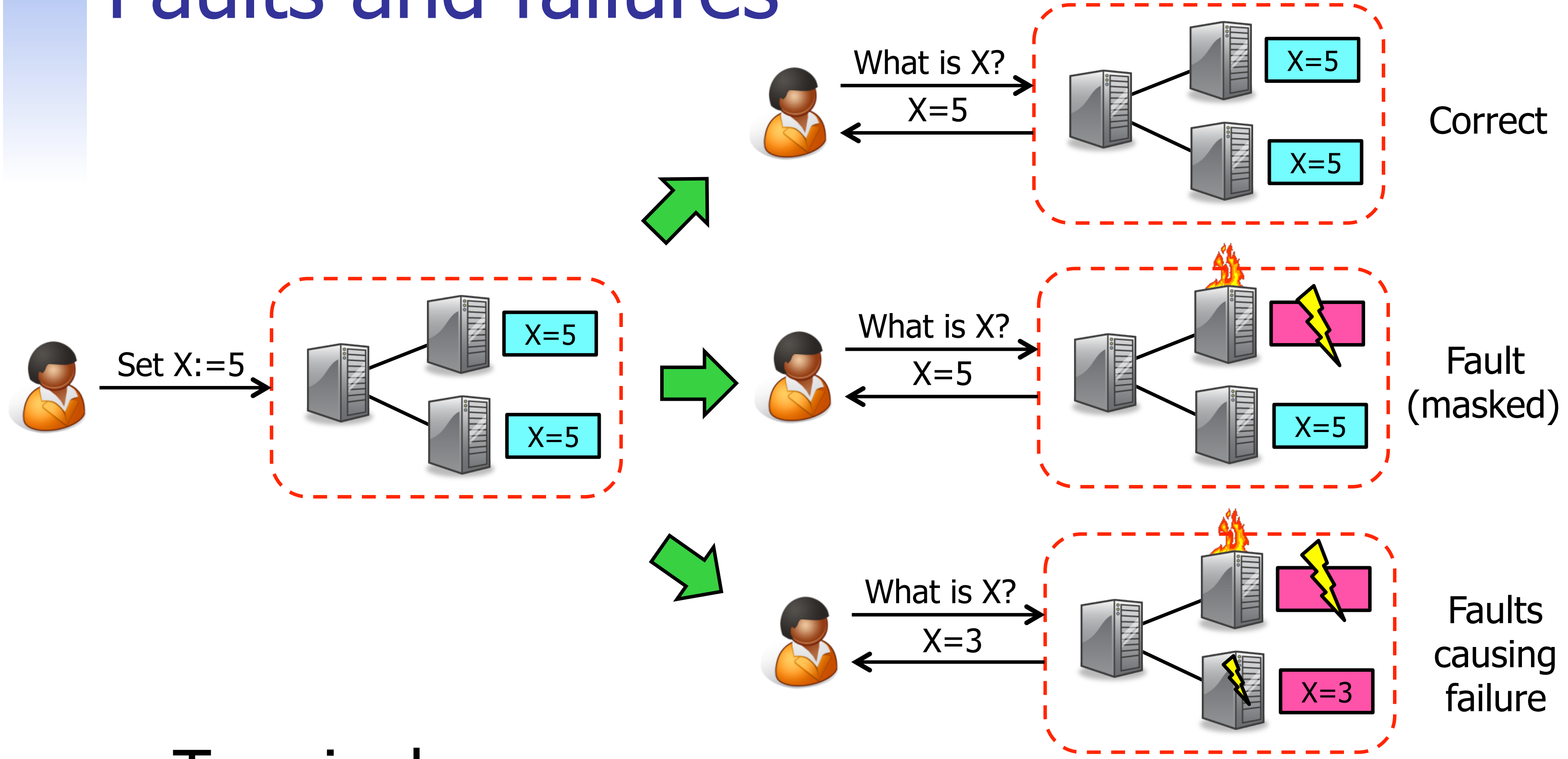
Plan for today

- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Synchronization, consistency ✓
  - Architectures: SMP, NUMA, Shared-nothing ✓
- Wide-area network ✓
  - Latency, packet loss, bottlenecks, and why they matter ✓
- Distributed programming and its challenges
  - Faults, failures, and what we can do about them NEXT
  - Network partitions, CAP theorem, relaxed consistency

Complications in wide-area networks

- Communication is slower, less reliable
  - Latencies are higher, more variable
  - Bottleneck capacity is lower
  - Packet loss, reordering, queueing delays
- Faults are more common
  - Broken or malfunctioning nodes
  - Network partitions

Faults and failures



- Terminology:
  - Fault: Some component is not working correctly
  - Failure: System as a whole is not working correctly

Faults in distributed systems

- What could possibly go wrong?
  - Node loses power
  - Hard disk fails
  - Administrator accidentally erases data
  - Administrator configures node incorrectly
  - Software bug triggers
  - Network overloaded, drops lots of packets
  - Hacker breaks into some of the nodes
  - Disgruntled employee manipulates node
  - Fire breaks out in data center where node resides
  - Police confiscates node because of illegal activity
  - …

Common misconceptions about faults

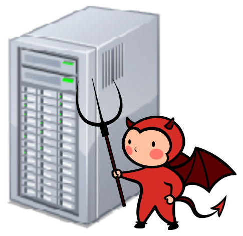
- "Faults are rare exceptions"
  - NO! At scale, faults are occurring all the time
  - Stopping the system while handling the fault is NOT an option – system needs to continue despite the fault
- "Faulty machines always stop/crash"
  - NO! There are many types of faults with different effects
  - If your system is designed to handle only crash faults and another type of fault occurs, things can become very bad

Types of faults

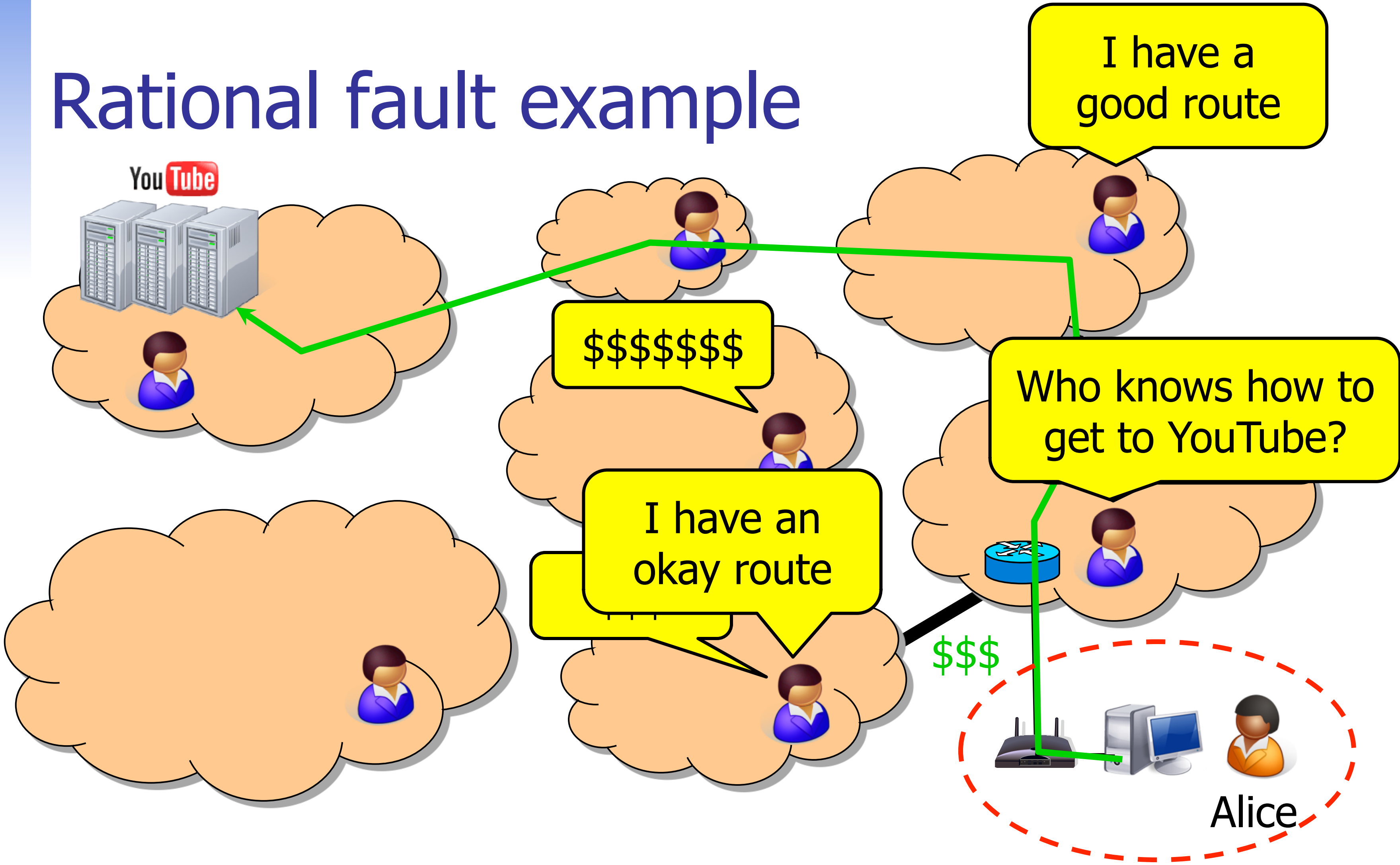
- Crash faults
  - Node simply stops
  - Examples: OS crash, power loss

- Rational behavior
  - Owner manipulates node to increase profit
  - Example: Traffic attraction attack (see next slide)

- Byzantine faults
  - Arbitrary - faulty node could do anything (stop, tamper with data, tell lies, attack other nodes, send spam, spy on user...)
  - Example: Node compromised by a hacker, data corruption, hardware defect...



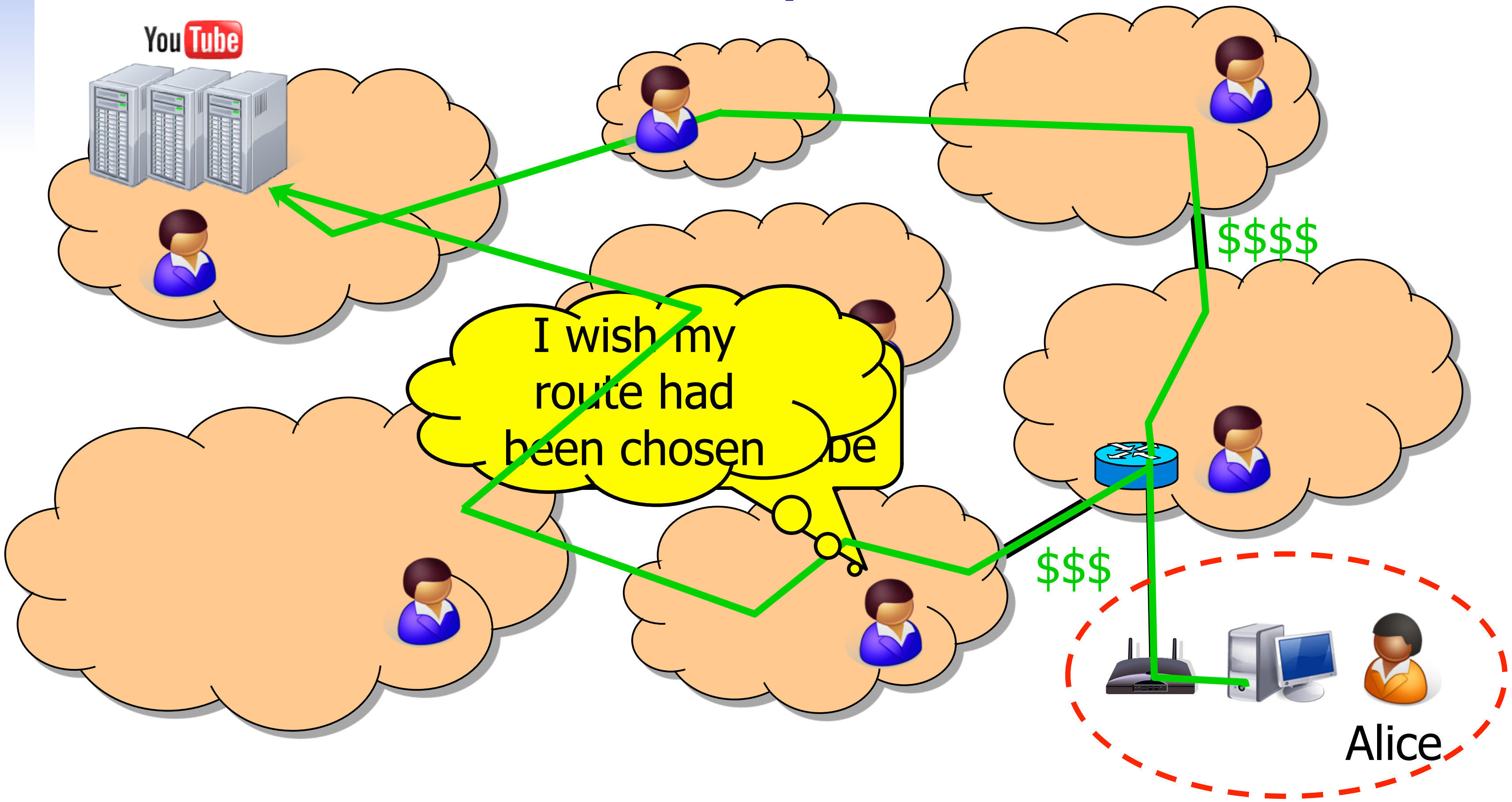
Rational fault example



- Alice's provider can choose between several routes to the same destination

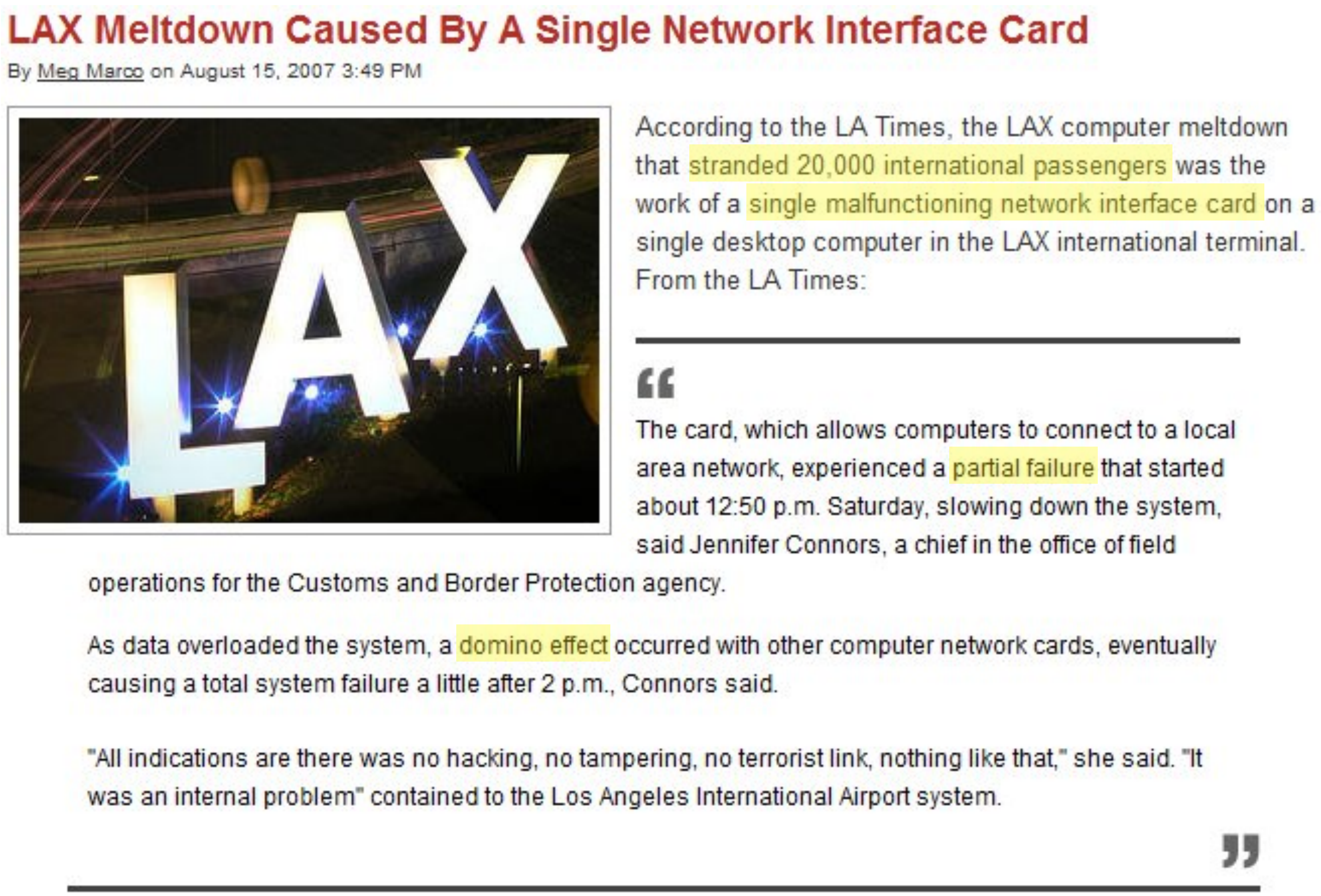
Traffic attraction: see Goldberg et al., "Rationality and Traffic Attraction: Incentives for honestly announcing paths in BGP", SIGCOMM 2010

Rational fault example



- Networks have an incentive to make their routes appear better than they are

Some examples of Byzantine faults



LAX Meltdown Caused By A Single Network Interface Card

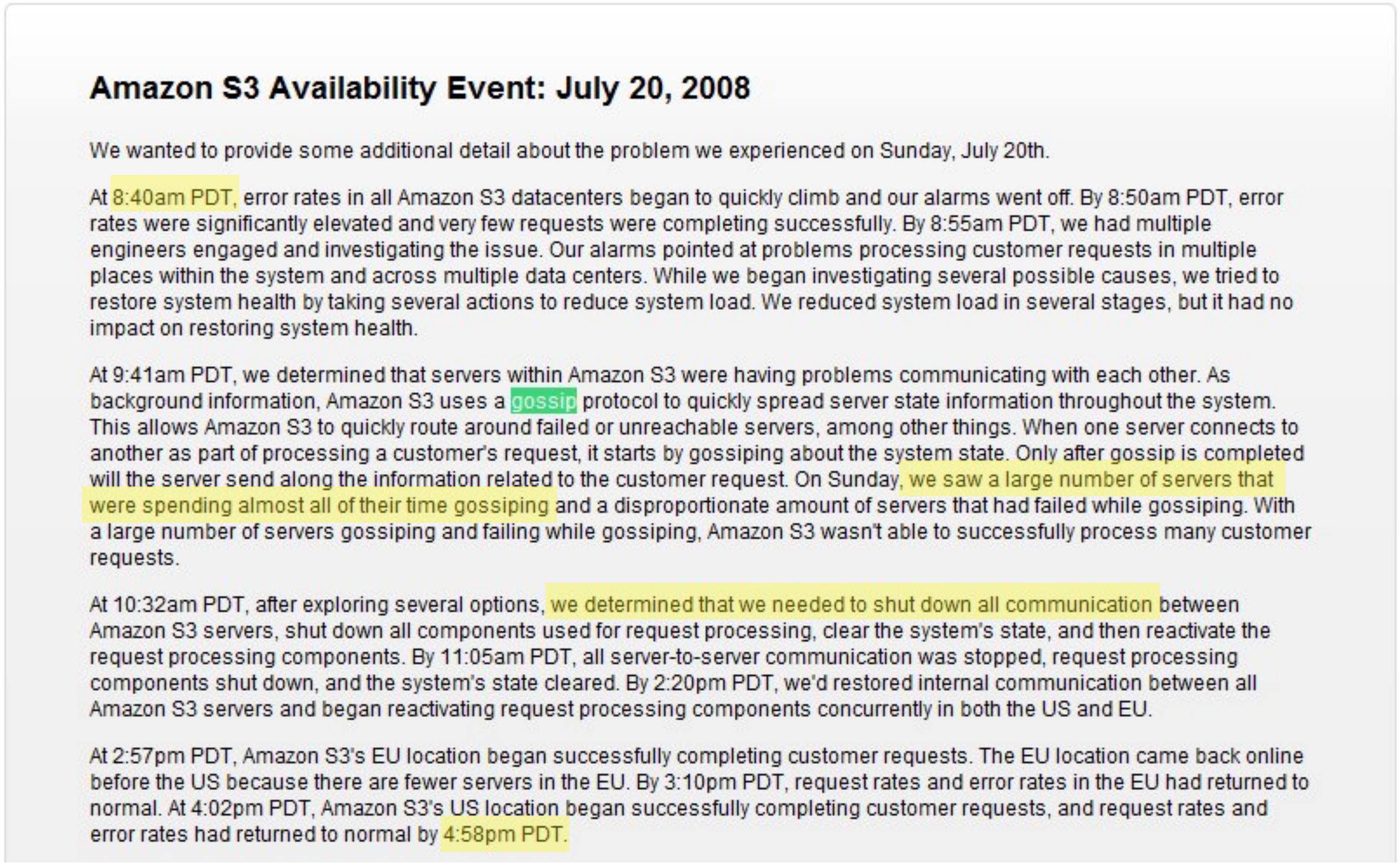
By Meg Marco on August 15, 2007 3:49 PM

According to the LA Times, the LAX computer meltdown that stranded 20,000 international passengers was the work of a single malfunctioning network interface card on a single desktop computer in the LAX international terminal. From the LA Times:

> The card, which allows computers to connect to a local area network, experienced a partial failure that started about 12:50 p.m. Saturday, slowing down the system, said Jennifer Connors, a chief in the office of field operations for the Customs and Border Protection agency.
>
> As data overloaded the system, a domino effect occurred with other computer network cards, eventually causing a total system failure a little after 2 p.m., Connors said.
>
> "All indications are there was no hacking, no tampering, no terrorist link, nothing like that," she said. "It was an internal problem" contained to the Los Angeles International Airport system.

http://consumerist.com/2007/08/lax-meltdown-caused-by-a-single-network-interface-card.html

Some examples of Byzantine faults



Amazon S3 Availability Event: July 20, 2008

We wanted to provide some additional detail about the problem we experienced on Sunday, July 20th.

At 8:40am PDT, error rates in all Amazon S3 datacenters began to quickly climb and our alarms went off. By 8:50am PDT, error rates were significantly elevated and very few requests were completing successfully. By 8:55am PDT, we had multiple engineers engaged and investigating the issue. Our alarms pointed at problems processing customer requests in multiple places within the system and across multiple data centers. While we began investigating several possible causes, we tried to restore system health by taking several actions to reduce system load. We reduced system load in several stages, but it had no impact on restoring system health.

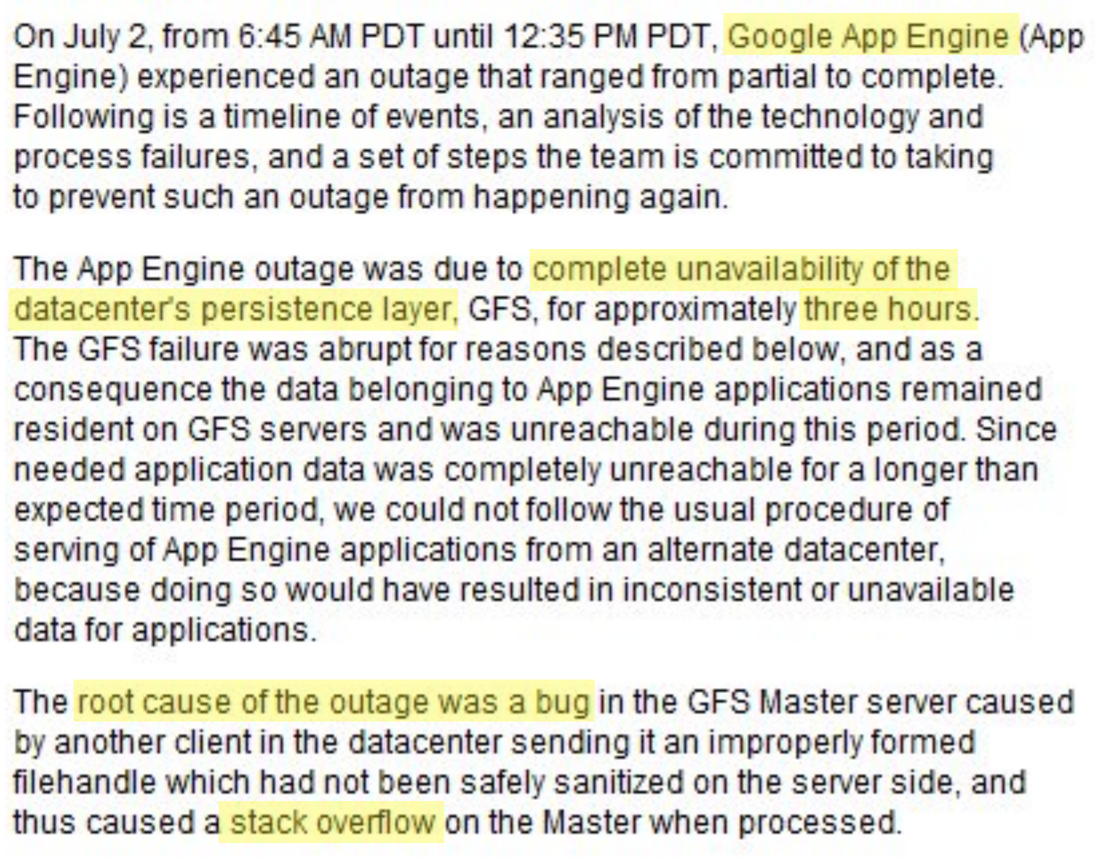
At 9:41am PDT, we determined that servers within Amazon S3 were having problems communicating with each other. As background information, Amazon S3 uses a gossip protocol to quickly spread server state information throughout the system. This allows Amazon S3 to quickly route around failed or unreachable servers, among other things. When one server connects to another as part of processing a customer's request, it starts by gossiping about the system state. Only after gossip is completed will the server send along the information related to the customer request. On Sunday, we saw a large number of servers that were spending almost all of their time gossiping and a disproportionate amount of servers that had failed while gossiping. With a large number of servers gossiping and failing while gossiping, Amazon S3 wasn't able to successfully process many customer requests.

At 10:32am PDT, after exploring several options, we determined that we needed to shut down all communication between Amazon S3 servers, shut down all components used for request processing, clear the system's state, and then reactivate the request processing components. By 11:05am PDT, all server-to-server communication was stopped, request processing components shut down, and the system's state cleared. By 2:20pm PDT, we'd restored internal communication between all Amazon S3 servers and began reactivating request processing components concurrently in both the US and EU.

At 2:57pm PDT, Amazon S3's EU location began successfully completing customer requests. The EU location came back online before the US because there are fewer servers in the EU. By 3:10pm PDT, request rates and error rates in the EU had returned to normal. At 4:02pm PDT, Amazon S3's US location began successfully completing customer requests, and request rates and error rates had returned to normal by 4:58pm PDT.

http://status.aws.amazon.com/s3-20080720.html

Some examples of Byzantine faults



On July 2, from 6:45 AM PDT until 12:35 PM PDT, Google App Engine (App Engine) experienced an outage that ranged from partial to complete. Following is a timeline of events, an analysis of the technology and process failures, and a set of steps the team is committed to taking to prevent such an outage from happening again.

The App Engine outage was due to complete unavailability of the datacenter's persistence layer, GFS, for approximately three hours. The GFS failure was abrupt for reasons described below, and as a consequence the data belonging to App Engine applications remained resident on GFS servers and was unreachable during this period. Since needed application data was completely unreachable for a longer than expected time period, we could not follow the usual procedure of serving of App Engine applications from an alternate datacenter, because doing so would have resulted in inconsistent or unavailable data for applications.

The root cause of the outage was a bug in the GFS Master server caused by another client in the datacenter sending it an improperly formed filehandle which had not been safely sanitized on the server side, and thus caused a stack overflow on the Master when processed.

http://groups.google.com/group/google-appengine/msg/ba95ded980c8c179

Correlated faults

- A single problem can cause many faults
  - Example: Overloaded machine crashes, increases load on other machines → domino effect
  - Example: Bug is triggered in a program that is used on lots of machines
  - Example: Hacker manages to break into many computers due to a shared vulnerability
  - Example: Machines may be connected to the same power grid, cooled by the same A/C, managed by the same admin
  - ...
- Why is this problematic?

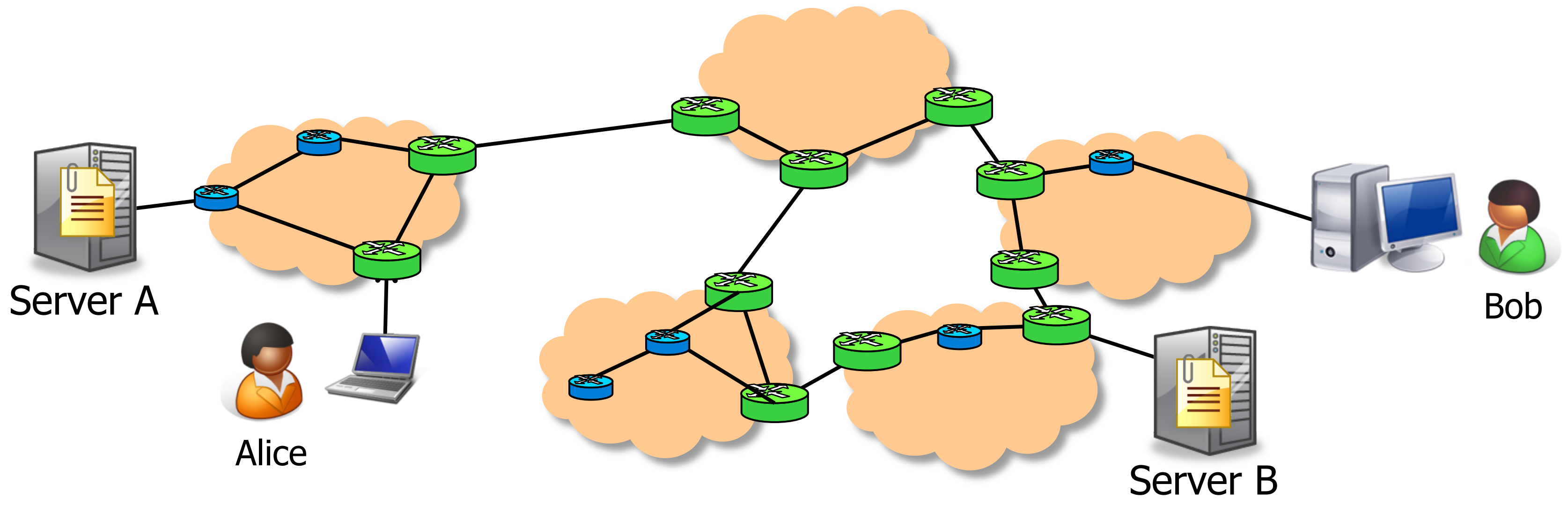
Recap: Faults and failures

- Faults happen all the time
  - Hardware malfunction, software bug, manipulation, hacker break-ins, misconfiguration, ...
  - NOT a rare occurrence at scale – must design system to handle them
- All faults are NOT independent crash faults
  - Faults can be correlated
  - Rational and Byzantine faults are real
- Three common fault models:
  - Crash fault model: Faulty machines simply stop
  - Rational model: Machines manipulated by selfish owners
  - Byzantine fault model: Faulty machines could do anything

What can we do?

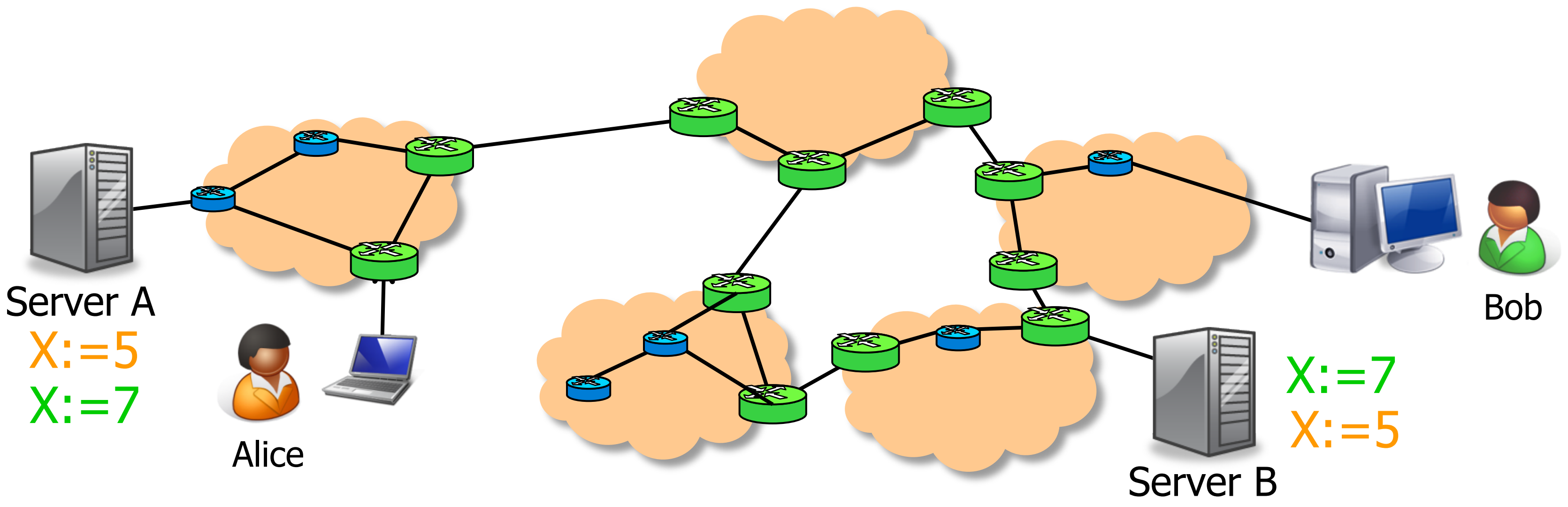
- Prevention and avoidance
  - Example: Prevent crashes with software verification
    - See formal methods at AWS: http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf
  - Example: Provide incentives for participation
- Detection
  - Example: Cross-check network's route announcements with other information to see whether it is lying, and hold it accountable if it is (e.g., sue for breach of contract)
- Masking
  - Example: Store replicas of the data on multiple nodes; if data is lost or corrupted on one of them, we still have the other copies
- Mitigation

Masking faults with replication



- Alice can store her data on both servers
- Bob can get the data from either server
  - A single crash fault on a server does not lead to a failure
  - Availability is maintained
  - What about other types of faults, or multiple faults?

Problem: Maintaining consistency



- What if multiple clients are accessing the same set of replicas?
  - Requests may be ordered differently by different replicas
  - Result: Inconsistency! (remember race conditions?)
  - For what types of requests can this happen?
  - What do we need to do to maintain consistency?

Types of consistency

- Strong consistency
  - After an update completes, any subsequent access will return the updated value
- Weak consistency
  - Updated value not guaranteed to be returned immediately, only after some conditions are met (inconsistency window)
- Eventual consistency
  - A specific type of weak consistency
  - If no new updates are made to the object, eventually all accesses will return the last updated value

Eventual consistency variations

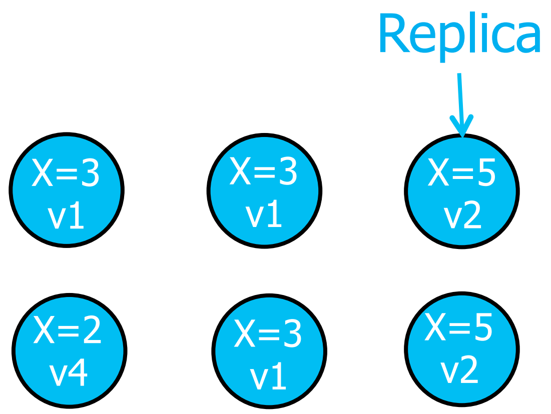
- Causal consistency
  - If client A has communicated to client B that it has updated a data item, a subsequent access by B will return the updated value, and a write is guaranteed to supersede the earlier write. Client C that has no causal relationship to client A is subject to the normal eventual consistency rules
- Read-your-writes consistency
  - Client A, after it has updated a data item, always accesses the updated value and will never see an older value
- Session consistency
  - Like previous case but in the context of a session, for as long as the sessions remains alive

Eventual consistency variations

- Monotonic read consistency
  - If client A has has seen a particular value for the object, any subsequent accesses will never return any previous values
- Monotonic write consistency
  - In this case the system guarantees to serialize the writes by the same process
  - Systems that do not guarantee this level of consistency are notoriously hard to program
- Few consistency properties can be combined
  - monotonic reads + read-your-writes most desirable for eventual consistency. Why?

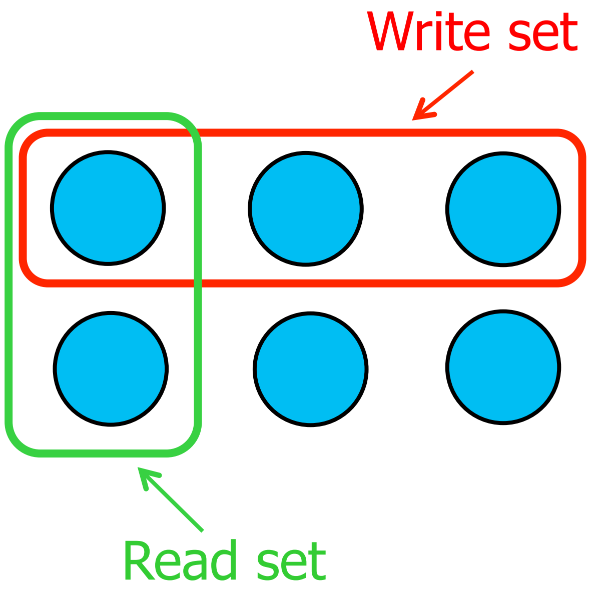
Example: Storage system

- Scenario: Replicated storage
  - We have N nodes that can store data
  - Data contains a monotonically increasing timestamp
- To write a value:
  - Pick W replicas and write the value to each, using a fresh timestamp (say, the current wallclock time)
- To read a value:
  - Pick R replicas and read the value from each
  - Return the value with the highest timestamp
  - If any replicas had a lower timestamp, send them the newer value



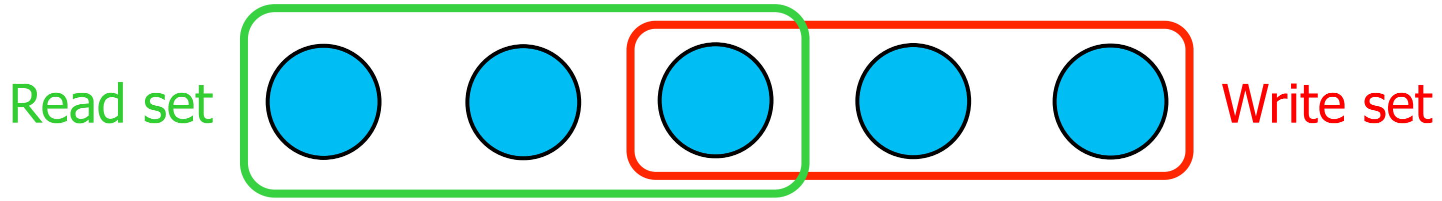
How to set N, R, and W

- For strong consistency?
  - What happens otherwise?
  - Will the data ever become consistent again?
- To avoid conflicting writes?
- To make reads fast? Writes?
- To minimize the risk of data loss?
- Let's do some examples!
  - N=2, W=2, R=1
  - N=2, W=1, R=1



Strong consistency: Quorum principle

- Majority quorum
  - Always write to and read from a majority of nodes
    - At least one node knows the most recent value
  - Pro: tolerate up to ⌈N/2⌉ - 1 crashes
  - Con: have to read/write ⌊N/2⌋ + 1 values

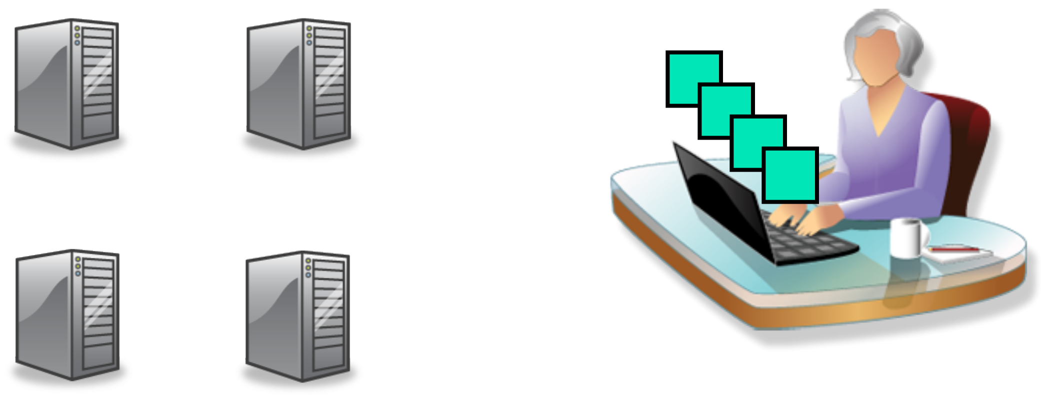


- Read/write quorums
  - Read R nodes, write W nodes, s.t. R + W > N
  - Pro: adjust performance of reads/writes
  - Con: availability can suffer

Consensus

- Replicas need to agree on a single order in which to execute client requests
  - How can we do this?
  - Does the specific order matter?
- Problem: What if some replicas are faulty?
  - Crash fault: Replica does not respond; no progress (bad)
  - Byzantine fault: Replica might tell lies, corrupt order (worse)
- Solution: Consensus protocol
  - Paxos (for crash faults), PBFT (for Byzantine faults)
  - Works as long as no more than a certain fraction of the replicas are faulty (PBFT: one third)

How do consensus protocols work?

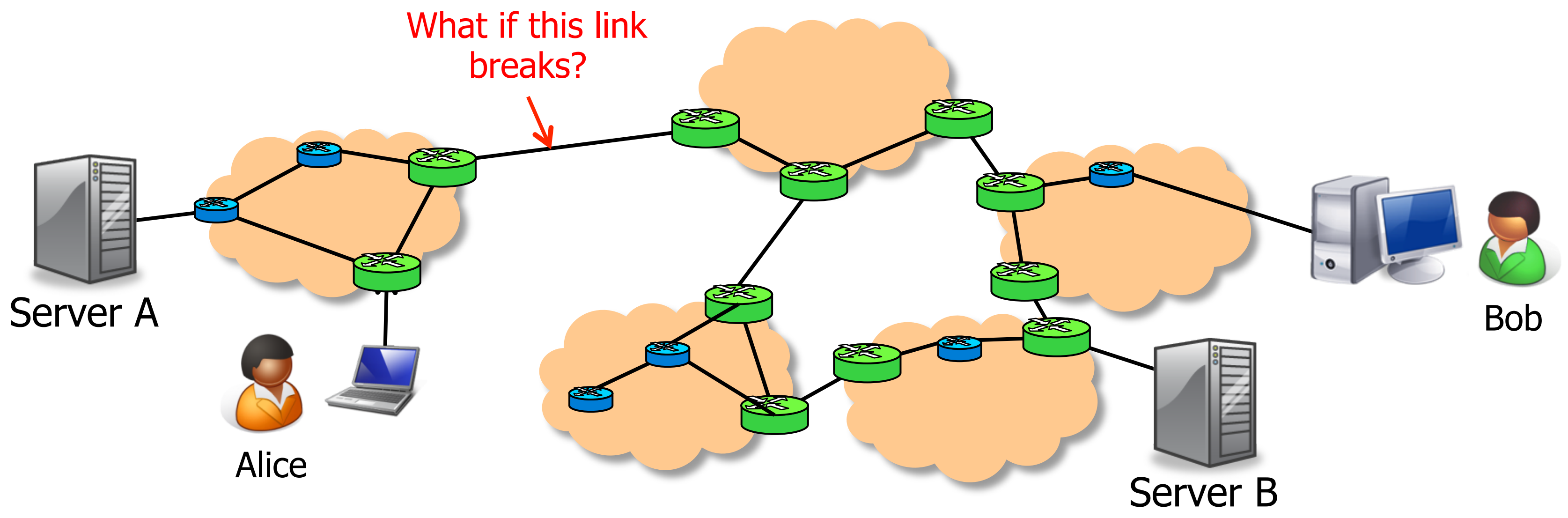


- Idea: Correct replicas 'outvote' faulty ones
  - Clients send requests to each of the replicas
  - Replicas coordinate and each return a result
  - Client chooses one of the results, e.g., the one that is returned by the largest number of replicas
  - If a small fraction of the replicas returns the wrong result, or no result at all, they are 'outvoted' by the other replicas

Plan for today

- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Synchronization, consistency ✓
  - Architectures: SMP, NUMA, Shared-nothing ✓
- Wide-area network ✓
  - Latency, packet loss, bottlenecks, and why they matter ✓
- Distributed programming and its challenges
  - Faults, failures, and what we can do about them ✓
  - Network partitions, CAP theorem, relaxed consistency NEXT

Network partitions



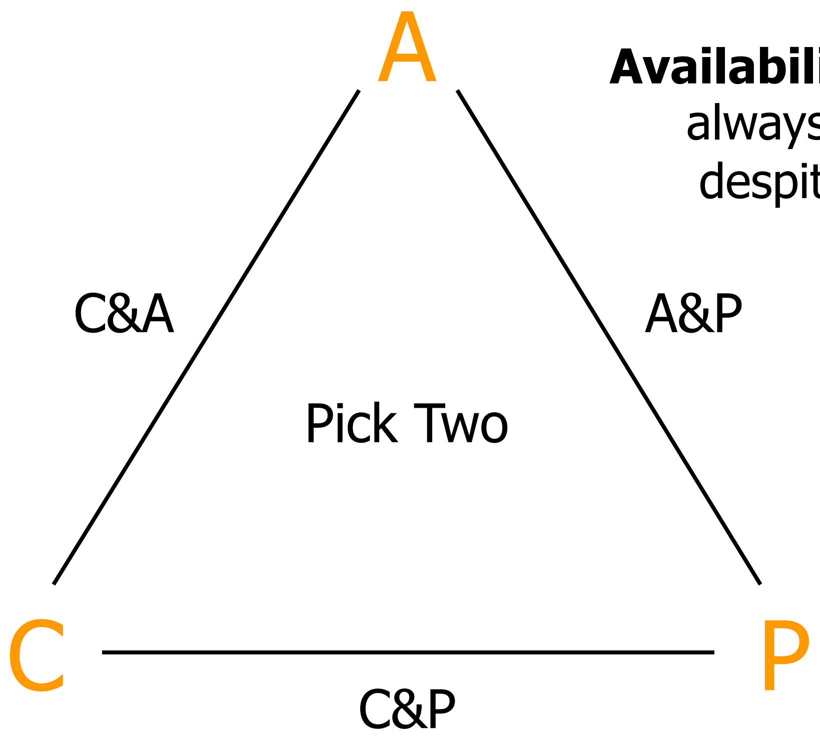
- Network can partition
  - Hardware fault, router misconfigured, undersea cable cut, ...
  - Result: Gobal connectivity is lost
  - What does this mean for the properties of our system?

The CAP theorem

- What we want from a web system:
  - Consistency: All clients single up-to-data copy of the data, even in the presence of concurrent updates
  - Availability: Every request (including updates) received by a non-failing node in the system must result in a response, even when faults occur
  - Partition-tolerance: Consistency and availability hold even when the network partitions

- Can we get all three?
  - CAP theorem: We can get at most two out of the three
    - Which ones should we choose for a given system?
  - Conjecture by Brewer; proven by Gilbert and Lynch

Visual CAP



Availability: Each client can always read and write despite node failures

Consistency: All clients always have the same view of the data at the same time

Partition-tolerance: The system continues to operate despite arbitrary message loss

Common CAP choices

- Example #1: Consistency & Partition tolerance
  - Many replicas + consensus protocol
  - Do not accept new write requests during partitions
  - Certain functions may become unavailable

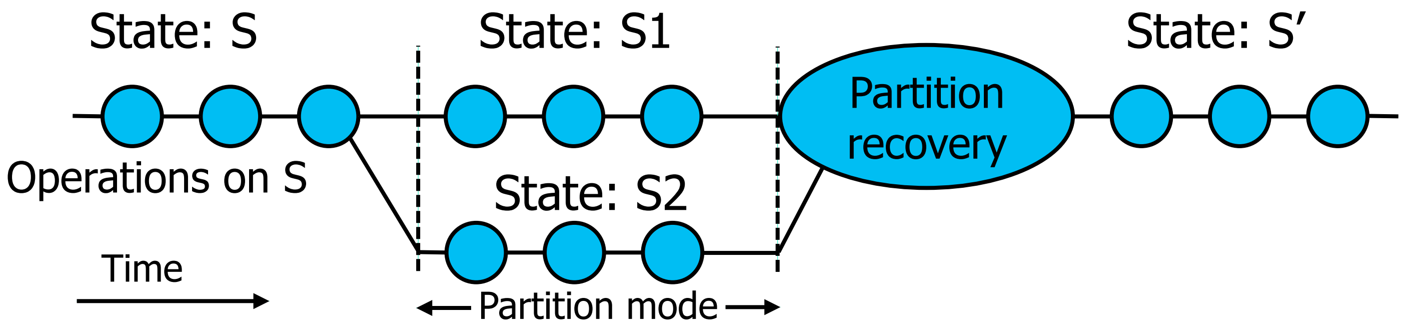
- Example #2: Availability & Partition tolerance
  - Many replicas + relaxed consistency
  - Continue accepting write requests
  - Clients may see inconsistent state during partitions

“2 of 3” view is misleading

- Meaning of C&A over P is unclear
  - If a partition occurs, the choice must be reverted to C or A
  - No reason to forfeit C or A when system is not partitioned
- Choice of C and A can occur many times within the same system at fine granularity
- Three properties are more of a continuous
  - Availability is 0 to 100
  - Many levels of consistency
  - Disagreement within the system whether a partition exists
- The modern CAP goal should be to maximize application-specific combinations of C and A

Dealing with partitions

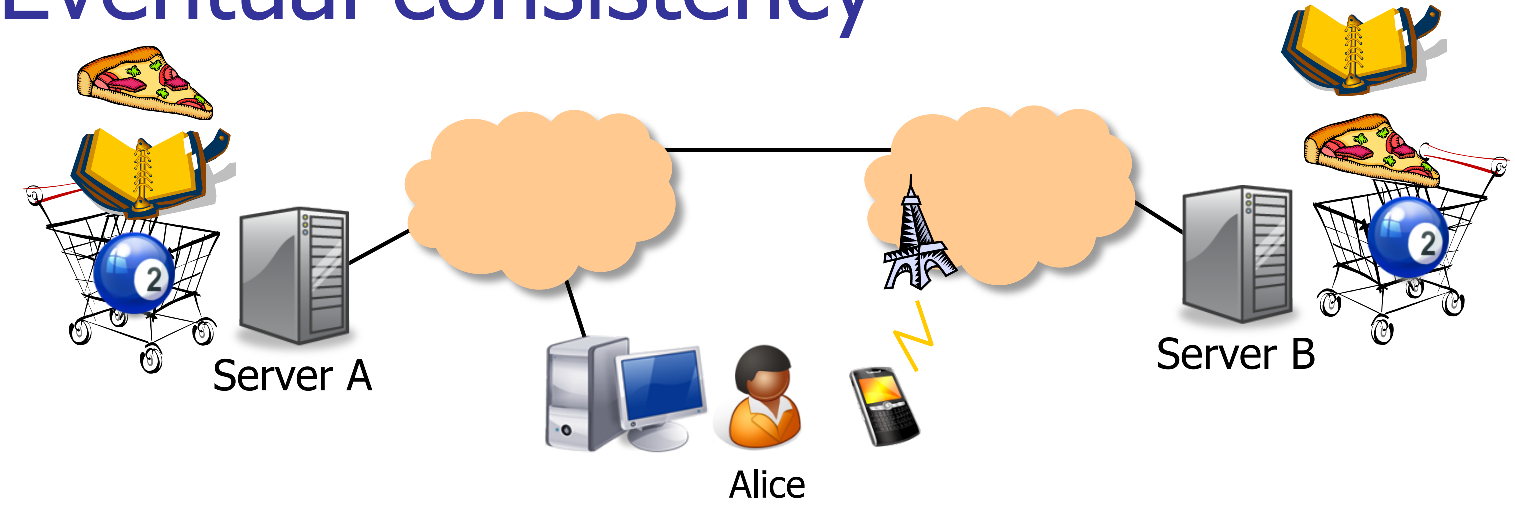
- Detect partition
- Enter an explicit partition mode that can limit some operations
- Initiate partition recovery when communication is restored
  - Restore consistency and compensate for mistakes made while the system was partitioned



Which operations should proceed?

- Depends primarily on the invariants that the system intends to maintain
- If an operation is allowed and turns out to violate an invariant, the system must restore the invariant during recovery
  - Example: 2 objects are added with the same (unique) key; to restore, we check for duplicate keys and merge objects
- If invariant cannot be violated, system must prohibit or modify the operation (e.g. record the intent and execute it after)
  - Example: delay charging the credit card; user does not see system is not available

Eventual consistency



- Idea: Optimistically allow updates
  - Don't coordinate with ALL replicas before returning response
  - But ensure that updates reach all replicas eventually
    - What do we do if conflicting updates were made to different replicas?
  - Good: Decouples replicas. Better performance, availability under partitions
  - (Potentially) bad: Clients can see inconsistent state

Partition recovery

- State on both sides must become consistent
- Compensation for mistakes during partition
- Start from state at the time of the partition and roll forward both sets of operations in some way, maintaining consistency
- The system must also merge conflicts
  - constraint certain operations during partition mode so that conflicts can always be merged automatically
  - detect conflicts and report them to a human
  - use commutative operations as a general framework for automatic state convergence
    - commutative replicated data types (CRDTs)

Compensate for mistakes

- Tracking and limitation of partition-mode operations ensures the knowledge of which invariants could have been violated
  - trivial ways such as "last writer wins", smarter approaches that merge operations, and human escalation
- For externalized mistakes typically requires some history about externalized outputs
- System could execute orders twice
  - If the system can distinguish two intentional orders from two duplicate orders, it can cancel one of the duplicates
  - If externalized, send an e-mail explaining the order was accidentally executed twice but that the mistake has been fixed and to attach a coupon for a discount

Relaxed consistency: ACID vs. BASE

- Classical database systems: ACID semantics
  - Atomicity
  - Consistency
  - Isolation
  - Durability

- Modern Internet systems: BASE semantics
  - Basically Available
  - Soft-state
  - Eventually consistent

Recap: Consistency and partitions

- Use replication to mask limited # of faults
  - Can achieve strong consistency by having replicas agree on a common request ordering
  - Even non-crash faults can be handled, as long as there are not too many of them (typical limit: 1/3)

- Partition tolerance, availability, consistency?
  - Can't have all three (CAP theorem)
  - Typically trade-off between C and A
  - If service works with weaker consistency guarantees, such as eventual consistency, can get a compromise (BASE)

Plan for today

- Parallel programming and its challenges ✓
  - Parallelization and scalability, Amdahl's law ✓
  - Synchronization, consistency ✓
  - Architectures: SMP, NUMA, Shared-nothing ✓
- Wide-area network ✓
  - Latency, packet loss, bottlenecks, and why they matter ✓
- Distributed programming and its challenges ✓
  - Faults, failures, and what we can do about them ✓
  - Network partitions, CAP theorem, relaxed consistency ✓

Stay tuned

Next time you will learn about:

Cloud basics

Assigned reading: “BASE: An ACID Alternative” by Dan Pritchett, http://queue.acm.org/detail.cfm?id=1394128