6.033: Computer System Engineering

Rachel Wu

Spring 2017

These are my lecture notes from 6.033, Computer System Engineering, at the Massachusetts Institute of Technology, taught this semester (Spring 2017) by Professor Katrina LaCurts[1]. My recitation instructor was Professor Michael Yee[2].

This semester, I could not make half the 6.033 lectures, so half the notes are summaries and paraphrases of associated book readings.[3] The other half I wrote in LaTeX in real time during class, so there may be errors and typos. In addition, I have included relevant discussions from recitation and tutorial. I have lovingly pillaged Tony Zhang's[4] and Evan Chen's formatting commands. Should you encounter an error in the notes, wish to suggest improvements, or alert me to a failure on my part to keep the web notes updated, please contact me at rmwu@mit.edu.

This document was last modified 2017-05-15.

[1] lacurts@mit.edu
[2] myee@mit.edu
[3] Jerome H. Saltzer, M. Frans Kaashoek. Principles of Computer System Design. 2009.
[4] txz@mit.edu

Contents

1 February 7, 2017 (R) 1
- 1.1 Administrivia . . . 1
- 1.2 Introduction . . . 1

2 February 8, 2017 (L) 2
- 2.1 Complexity in systems . . . 2
- 2.2 Addressing complexity . . . 2
- 2.3 Computer systems . . . 3
- 2.4 Client-service organization . . . 3

3 February 10, 2017 (T) 5
- 3.1 Fridays in 6.033 . . . 5
- 3.2 Communication basics . . . 5

4 February 13, 2017 (L) 6

5 February 14, 2017 (R) 6
- 5.1 Naming . . . 6
- 5.2 DNS . . . 6

6 February 15, 2017 (L) 8
- 6.1 Virtualization . . . 8
- 6.2 Modularity in memory . . . 8
- 6.3 Virtual memory . . . 8

7 February 16, 2017 (R) 9
- 7.1 UNIX . . . 9

8 February 17, 2017 (T) 9
- 8.1 Writing a system critique . . . 9

9 February 21, 2017 (L) 10
- 9.1 Bounded buffers . . . 10
- 9.2 Concurrency . . . 10
- 9.3 Locks . . . 10

10 February 22, 2017 (L) 12
- 10.1 Threads . . . 12
- 10.2 Scheduling . . . 12

11 February 23, 2017 (R) 13
- 11.1 UNIX discussion . . . 13

12 February 24, 2017 (T) **14**
12.1 Design project overview 14

13 February 27, 2017 (L) **15**
13.1 Kernels 15
13.2 Virtual machines 15

14 February 28, 2017 (R) **17**
14.1 Eraser discussion 17

15 March 1, 2017 (L) **18**
15.1 Performance 18
15.2 Layering memory 19

16 March 2, 2017 (R) **20**
16.1 Map-Reduce discussion 20

17 March 6, 2017 (L) **21**
17.1 Introduction to networking 21
17.2 History of the Internet 21

18 March 7, 2017 (R) **23**
18.1 DARPA Internet discussion 23

19 March 8, 2017 (L) **24**
19.1 Routing protocols 24
19.2 Autonomous systems 24
19.3 BGP 25

20 March 9, 2017 (R) **26**
20.1 Resilient overlay networks 26
20.1.1 Failure detection and correction 26
20.1.2 Application integration 27
20.1.3 Expressive policy routing 27

21 March 13, 2017 (L) **28**
21.1 Reliable transport 28
21.2 Congestion control 29

22 March 16, 2017 (R) **30**
22.1 Bufferbloat discussion 30

23 March 20, 2017 (L) **31**
23.1 File sharing 31
23.2 BitTorrent 31
23.3 VoIP 31

24 March 21, 2017 (R) **32**
24.1 Data center TCP discussion 32

25 March 23, 2017 (R) **33**
25.1 Content delivery networks discussion (Akamai) 33

26 April 3, 2017 (L) **34**
26.1 Fault tolerance 34

27 April 10, 2017 (L) **36**
27.1 Atomicity 36
27.2 Write-ahead logging 36

28 April 24, 2017 (L) **37**
28.1 Availability through replication 37

29 May 1, 2017 **39**
29.1 Principal authentication 39

30 May 15, 2017 **41**
30.1 Ransomware 41
30.2 Anonymity 41

1 February 7, 2017 (R)

1.1 Administrivia

- Lectures in 26-100 on Mondays, Wednesdays. Recitations on Tuesdays, Thursdays. Tutorials on Fridays. Fill out the form to sign up for times.
- Prerequisites: 6.004.
- Two quizzes (one is final exam).
- Hands-on due every Tuesday. Readings due for recitations.
- Design project, with various stages throughout the semester.

1.2 Introduction

This class is about building large systems and discussing the tradeoffs associated with design choices.

2 February 8, 2017 (L)

2.1 Complexity in systems

A system is a conglomeration of many parts that work together, so there are many factors that can introduce complexity. These factors include:

- Some issues only appear when all the parts of a system are assembled. These are known as **emergent properties**, or surprises.
- In a large system, **propagation of effects** can cause large repercussions from small changes.
- As systems increase in speed and size, different elements scale at different rates, known as **incommensurate scaling**. For example, we cannot simply build a larger pyramid by scaling it up equally in all dimensions; this is because the weight of a pyramid increases cubically as size, while the strength of stone only increases quadratically as size.
- There are limited resources and unlimited desires, a common theme in economics, as well as systems. In order to 1) maximize utility, 2) avoid wasting resources, and 3) allocate most efficiently, we have to make **tradeoffs**. For example, in binary classification, we make tradeoffs between misclassification and false-positive rate, often by using a **proxy**, or alternative metric.

Complex systems interface with their environments. When studying system, we consider the **purpose** and **granularity** with which to view the system. For example, a student might view his dormitory as a system of social interactions, while an civil engineer views it as a building to be maintained. In addition, we can abstract out **sub-systems** and change the granularity with which we consider systems and their components. Complexity also arises from 1) many components (nodes), 2) many connections (edges), 3) inconsistencies, 4) many specifications, and 5) requiring many different human resources.

Dependencies are the worst, and complexity grows exponentially as requirements. However, we should also avoid excessive generality.

> "If it is good for everything, it is good for nothing"—pcsd (16)

However, requirements change, so as we make lots of minor changes instead of redesigning the system, the minor changes accumulate and add complexity. We should consider the law of diminishing returns when adding features and fixes.

2.2 Addressing complexity

There are several ways to reduce or cope with complexity. These include:

- Separate interface from implementation with **abstractions**. Abstraction by natural boundaries is known as **functional modularity**. One way to abstract is by **naming**, which simply refers to a module, irrespective of its implementation.

- Be **fault tolerant** by accepting many inputs and being strict on outputs. Take reasonable inputs, but immediately report wrong ones.
- **Layering** helps modularity. For example, only allow a module to talk to others in its layer, and its counterparts in adjacent layers.
- Tree-like structures, known as **hierarchies**, help organize systems.

2.3 Computer systems

Computer systems are similar to most systems, except software allows us to surpass most physical limitations, and they change at unprecedented rates. The static discipline guarantees inputs and outputs, so software simply works. We are limited not by how things fit together, but by how complicated a system the human can understand. In addition, computer systems change so quickly that we often do not fine-tune them, as we would a building or spaceship; we simply build newer, better ones.

However, it is hard to design a system in the *right* ways, even if we do have general design principles. Therefore, design happens in **iterations**. It is good practice to 1) take small steps, 2) design with patience, 3) improve with feedback, and 4) study failures.

> "Complex systems fail for complex reasons"—pcsd (37)

Therefore, we should aim for simplicity.

2.4 Client-service organization

One method to enforce modularity is with the client-service model. In particular, the only form of communication between modules is **messages**. Using messages is modular for the implementer, fault tolerant for the system, and secure against attacks. The two most common methods of message passing are **remote procedure call** (RPC) and **publication-subscription** (pub-sub).

Soft modularity is when modularity is defined, but not enforced. This includes the **stack discipline**, which provides a rule for who should edit the stack, and how, but does not enforce that the rules are actually followed. In contrast, **enforced modularity** uses external mechanisms to integrate modularity into the nature of the system.

In the client-service organization, the **client** initiates requests, and the **service** responds. The process of **marshaling** converts objects to byte arrays, and unmarshaling reverses the process.

Slight disadvantages to this model are the speed and costs of maintaining distinct services and clients, but these are tradeoffs to be made (and it is possible to have a client and service on the same machine). However, there are many advantages:

- The client and service do not share state, other than messages.
- Error propagation is limited.
- Clients can protect themselves against service failures by setting a timeout limit.

- This model naturally leads to clear interfaces.

Clients and services can be 1 to n, n to 1, or n to n. One module can take on an arbitrary number of roles. Often, however, a **trusted intermediary** takes care of sending messages. This intermediary is especially effective in abstracting out to whom a message should be sent.

Example 2.1

When emailing out to a mailing list, you don't care who's on the mailing list; you simply care that it reaches the intended recipients, so the conduit takes care of that.

In RPCs, a **stub** is an abstraction that handles marshaling and low-level communication. RPCs are intended to emulate procedure calls, but they have the following (additional) differences.

- RPCs introduce the **no-response** failure. This is countered by **at-least once** RPC (try until it works), **at-most-once** RPC (error if fail), and **exactly-once** RPC (idealistically perfect).
- Features such as global variables are not effective through RPC protocols.

In addition, there are two ways to communicate. A **push** is when data is sent up, and a **pull** is when data is retrieved from the service. This is also an example of pub-sub.

Example 2.2

Email uses both pushing and pulling. The sender with SMTP pushes mail up, while the receiver pulls mail into their inbox.

3 February 10, 2017 (T)

3.1 Fridays in 6.033

Communication instructor Janis Melvold.[5]

- There will be 11 tutorials, both for communication and general review.
- The design project has a preliminary report, presentation, final report, and peer review.
- We will write 2 critiques as well.

3.2 Communication basics

A **genre** is a category or type of written communication. It has specific stylistic/structural features for a specific audience in context. For example, oral genres include lectures, eulogies, or TED talks. Written genres include a lab report, concert review, or even Tweet.

Sometimes it's hard to determine whether something is prior knowledge. Your audience could get confused, or feel patronized. Papers we read are written for specialists in the field, so we'll be confused!

New shiny too! We introduce the **what-how-why** system. For example,we ask "what is the system? how does it work? why were these decisions made?"

[5] melvoid@mit.edu

4 February 13, 2017 (L)

Lecture cancelled due to "snow." Notes merged with recitation notes the next day.

5 February 14, 2017 (R)

5.1 Naming

Names are used for modularity. For example, there is the DNS system, or class/function names, or file systems. They are user-friendly, allow indirection (can swap out binding to name later, indirection), can be used to retrieve and share objects, hiding modules' implementation, access control (e.g. GoogleDocs). Names allow us to view documents as objects.

In many systems, we can pass by value or reference. Reference is just a handle, or a name. Some names have structure, which allows them to be used as addresses. These could include where an object is, and how find it. There are three components to a naming scheme.

1. Name space.
2. Universe of values.
3. Look-up algorithm that maps names to values (resolution), given a context.

Example 5.1

For example, consider a file system. The names are the hierarchies of directories and the file. The values are the files themselves. There is some algorithm that maps paths to content, where context could be the current working directory or the root directory (for full paths).

Names could expire, or last forever (**stable binding**). DNS names can change, but some names can't change (e.g. registers). Not all names in the name space have to be bound, so we may encounter *not-found* results. Some naming schemes also support **reverse lookup**, where we provide a value and obtain the name

5.2 DNS

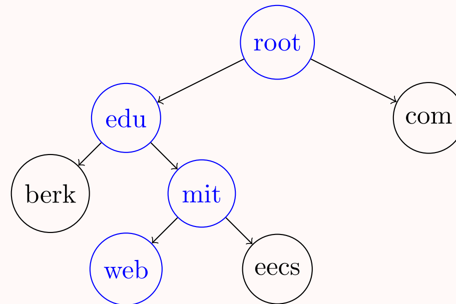
DNS, or domain name system, maps domain names to IP addresses. Any number of names can map to any number of IP addresses. You can repurpose both machines and names. Why do we use DNS?

- More user-friendly than 8-bit numbers.
- Hierarchy for decentralization.
- Easy to update, change, and scale.
- Robust performance; no performance bottleneck.

Distributed servers hold different hierarchies of bindings, and to find a binding, you have to search for it.

Example 5.2

Say we want to resolve `web.mit.edu`.



There are enhancements to DNS:

1. Ask anyone for initial request. Can just ask your local name server.
2. Recursion. Convenient, but not much faster than doing it yourself, thought it could be a little faster.
3. Caching. Search for something once, and you won't need to go back to the root for it next time.

We discuss benefits and drawbacks of DNS.

Benefits	Drawbacks
• scalable: hierarchy and distributed • management: naming distributed • performance: one-to-many, load balancing, caches • robust and reliable, with replicas and UDP	• malicious name servers (phishing attempts) • Great Firewall: map banned IPs to fake pages • could compromise an actual name server • could pretend to be a name server and return a fake response to someone • cache poisoning

6 February 15, 2017 (L)

6.1 Virtualization

Sometimes, we want to make the assumption that each application can run on its own virtual computer, through **virtualization**. **Multiplexing** is splitting one physical object to many virtual objects; **aggregating** is gluing together many physical objects into one virtual object. It makes no difference to the user; it feels as though there's just one entire machine; this is known as **emulation**.

> **Example 6.1**
> RAID is a method of virtualization by aggregation. RAM is virtualization through emulation.

We virtualize the processor through **threads**. Each thread knows the program counter and stack pointer (or equivalent). Often, programmers choose to use single-threaded programs, for simplicity, but it is also possible (generally) to run multiple threads concurrently, with the help of a **thread manager**. Interruptions can also be handled concurrently. **Exceptions** are interrupts that are relevant to the running thread.

Virtual memory is providing each thread with a fresh copy of its own memory, so they don't have to share. They also have **virtual address spaces**, so programs can be position independent (everyone can start at 0). Communication can occur through a **bounded buffer**, which is a blocking queue (which can be harder). Virtual machines are also helpful, especially for emulating hardware.

> **Example 6.2** (Hardware emulation)
> Suppose we want to design a new chip or piece of hardware. We can write an emulator in software, use that emulator to start developing programs for the new hardware, and then manufacture. Afterwards, the emulator can also help debug the hardware.

6.2 Modularity in memory

To enforce that shared physical memory is not corrupted by undesired programs, we introduce the idea of **domains**. The **memory manager** keeps track of the upper and lower bounds in memory, and access permissions. There are three types of permissions: read,write, and execute (rwx).[6] We can also "map" domains, to provide access to a domain to different threads.

6.3 Virtual memory

Memory management units, like page tables, map virtual addresses to physical addresses. All this was discussed in 6.004, so if you need a refresher, check out `http://6004.mit.edu`. Virtual memory is just a naming scheme, which makes our lives easier.

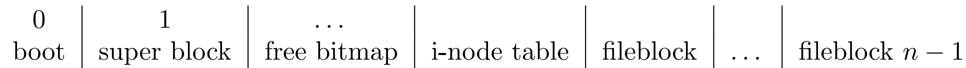
[6] In practice, the only useful combinations of permissions are r, rw, rx, rwx. Self-modifying programs are very dangerous, and can lead to loopholes for attacks.

7 February 16, 2017 (R)

7.1 UNIX

For this recitation, we read *The UNIX Time-Sharing System* by Ritchie and Thompson.

Names in UNIX include files, directories, I/O devices, users, user groups, etc. The general partition of a system looks like:



0 boot	1 super block	... free bitmap	i-node table	fileblock	...	fileblock $n-1$

The boot block contains vital information to boot up the OS. The super block contains information, such as size. The free blocks bitmap keeps track of which blocks are free. The i-node table keeps track of i-nodes; in particular, *root* is the first entry. We also discussed the various layers of the system.

	names	values	context	default/explicit
block layer	block #s	blocks	disk	def
file layer	block index	block #	i-node	expl
i-node #	i-node #	i-nodes	i-node table	def
filename	filename	i-node #	directory	expl
pathname	relative path	i-node #	current directory	expl
absolute path	abs path	i-node #	root	def

8 February 17, 2017 (T)

8.1 Writing a system critique

Questions we should ask are:

- what is my purpose?
- who am I writing for?
- how does that influence what I assume and explain?
- what is the system?
- how does it work?
- why does it work this way?

The notion of "stasis" in argumentation is an issue or question that must be resolved to advance an argument. Types include facts, definitions, causation, value, and action.

Example 8.1 (Murder, murder)

Hyde killed a person. We would ask, was it 1st degree, 2nd degree murder? Self defense? Why did Hyde kill someone?

9 February 21, 2017 (L)

I'm attending my first lecture live!

Our current goal is to enforce modularity on a single machine. In the previous lecture, we used virtual memory to separate programs' memory. Today, we will focus on how programs communicate through bounded buffers, and we still assume that there is one program per CPU.

programs shouldn't be able to refer to each other's memory.	$\rightarrow$	virtual memory
programs should be able to communicate.	$\rightarrow$	bounded buffer
programs should be able to share one CPU without disturbing others	$\rightarrow$	next time

9.1 Bounded buffers

Bounded buffers have two operations: **send** and **receive**. These are blocking buffers, but those are edge cases. Concurrency is problematic here. We cannot assume anything about interleaving, or even assume anything about a single line of code. In fact, something as simple as `x = x + 1` is compiled to around 3 lines of assembly, which could be interrupted at any time.

> "Concurrency will ruin your lives"—lacurts

We maintain an abstraction with n, the size of the buffer; in and out, the number of messages written to and read from the buffer.

- we are allowed to write when there is space, in $-$ out $< n$.
- we are allowed to read when there are messages, in $>$ out.
- we write to the out$\%n$ slot and read to the in$\%n$ slot.

9.2 Concurrency

Race conditions are bad! We need to change our buffer to support concurrency.

> "Today is class participation day. You are 'a' and 'b.' What are your names?" "Alanna." "Billy."

9.3 Locks

Locks allow one CPU to be inside a piece of code at a time. We can **acqure** and **release** a lock.[7]

[7] We do not acquire individual objects. Instead, we acquire *locks* to those objects.

Example 9.1 (Locks)
We have two CPUs talking to a bounded buffer.

```
send(msg)
1   write message to buffer
2   increment in count
```

CPU 1 sends 1, 2, and 3. CPU 2 sends 101, 102, and 103. If there are no locks, things mess up! If we lock every single line of code, things mess up again! Except we will stick everything in at least. We should lock at the beginning and end of the block in `send`. We want the `write` and `increment` to be atomic.

We then hit another issue: **deadlock**! But there is a way around it here: we can `release` the lock when we cannot send, and wait to `acquire` it again.

It's hard to decide where to put locks. Coarse-grained locks are easy to maintain correctness, but they will make things very slow, and we lose benefits of parallelism.

Example 9.2 (Filesystem lock)
Suppose we want to move a file. In theory, it should be easy.

```
move(dir1, dir2, filename)
1   unlink the file from dir1
2   link the file to dir2
```

If we lock the whole chunk, we couldn't move two unrelated files! So let's lock by directory. But what if the code were interrupted between the two lines? Inconsistent state exposed! But. . .

There are other ways out of deadlock. For example, we can impose **lock ordering**, so if there are conflicts, only one process can acquire the same lock at a time. This *does* require that there be a global ordering, which is not the most modular idea.

Locks can be implemented using hardware. There is an atomic operation called exchange (`XCHG`). At the lowest level, this is all hardware.

10 February 22, 2017 (L)

10.1 Threads

Today, we will virtualize the processor, so that many programs can run on the same CPU. The thread API has two operations, **resume** and **suspend**. We would like to design a thread manager that preserves modularity, so one thread does not cause other threads to die. Thread allocation happens in 3 steps: allocate memory, selects a processor, and sets the `PC` and `SP`.

Threads can give up their time by calling **yield**, which suspends the current thread, selects a new thread, and resumes that new thread. While threads may be written in any language, yield is generally low-level (SVC or similar). In particular, we maintain a processor table and a thread table. The **processor** table keeps track of which processor is currently running which thread, while the **thread table** keeps track of thread states.[8] Yield is important because most threads spend most of the time waiting for events to occur.

We also introduce the idea of interrupts and exceptions.

Remark 10.1. There are many, many words for slightly different, often conflicting meanings. Here, an **interrupt** has no relation to the current thread, and an **exception** has events related to the current thread.

It is a concern that an interrupt will attempt to yield a thread, after it has already acquired the lock to change thread and is trying to yield itself. Here, we will deadlock. Thus, we prevent this by disabling interrupts while a thread yields and switches to the new thread. The new thread can then be interrupted.

10.2 Scheduling

There are several models of scheduling.

1. The process described above is **non-preemptive scheduling**, where we simply wait for threads to yield. This is not effective since one thread could just endless loop, and the processor would be useless.

2. Some systems have **cooperative scheduling**, which means that all threads agree to play nice and yield every so often. This doesn't enforce anything, and isn't very effective.

3. Finally, we can force yielding through an interrupt through **preemptive scheduling**. The thread manager takes care of that.

We would also like to prevent excessive **polling**, or giving time to a thread that is not yet ready. This is a waste of processor time, and instead, we can introduce primitives to signal when a thread is ready to be run again. Naively, we could **notify** and **wait** for the notification, but we may miss the message. To augment, we could use event counts and sequences, which is a semaphore-like concept.

[8] We only need to keep track of the `SP` for each thread since the `PC` always points to where we suspended from, and threads share the same address space.

11 February 23, 2017 (R)

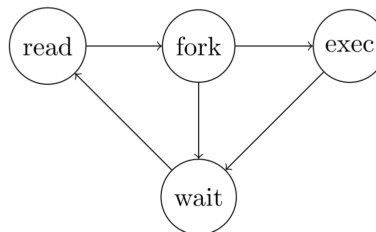
11.1 UNIX discussion

We review the UNIX paper. There were several true-false questions.

1. In UNIX, all processes receive equal fraction of CPU time. **False.**
2. 2 child processes can communicate via pipe only if the pipe was created by a common ancestor. **True.**
3. 2 processes can communicate via shared memory, regardless of whether they have a common ancestor. **False.** In this version of UNIX, there was no shared memory.
4. `execute()` creates a new process. **False.** It just overwrites the current process.
5. A pipe between two processes can't be established after both have started. **True.** In this version, the pipe is inherited when `fork()` spawns off child processes.

Common system calls include:

- `fork()` spins off a child thread and returns the process ID. A child knows that it's a child because the pid of the child is 0, and the pid of the parent is greater than 0. If a parent has a local variable, and the child changes it, nothing happens to the parent. In this version of UNIX, there are no file locks, so if the child inherits a copy of the parent's files, it could overwrite them.
- `exec()` just executes code.
- `wait()` can be used to coordinate between processes.



We simulate a simple shell. The REPL! tbt 6.037

```
1  while (1)
2      prompt();
3      read(cmd, args, background, infile);     // check for & to see if wait
4      pid=fork();
5      if (pid==0)                              // child process executes
6          if infile
7              close(0);                        // optionally redirect in
8              fd = open(infile);
9          exec(cmd, args)
10     else if not background wait();
```

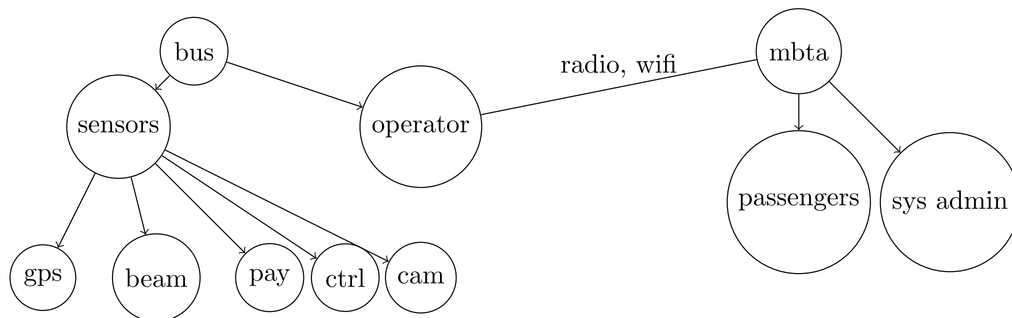
12 February 24, 2017 (T)

12.1 Design project overview

There are several goals of the system.

- assessing comfort, etc.
- addressing failure

Our system has several modules.



There are many modules that already exist and many to design ourselves.

Exists	Design	Tradeoffs
• bus routes, and how often to service them • sensors • availability of operators	• allocation of buses to routes • passenger feedback • detecting, addressing high demand • addressing failure • role of sys admins • sensor data collection • communication protocols • data storage on servers	• automation vs. human labor

13 February 27, 2017 (L)

"Thank you—ghost." —lacurts

13.1 Kernels

The **kernel** is a non-interruptible, trusted program that runs system code. We cannot enforce that the kernel is correct! In the unix/linux kernel, there is a lot of soft modularity, but it's mostly just a huge chunk of C.

Threads can only enter the kernel domain at **gates**, through supervisor calls (SVC) with the following procedure.

1. processor changes from user to kernel mode
2. PC set to gate entry point
3. kernel code is executed without interruption
4. processor changes from kernel to user mode
5. old PC loaded back

Kernel errors are *fatal*, so we try to limit the size of kernel code. It's terrible if an adversary can exploit bugs in the kernel! There are two models for kernels.

1. The **monolithic kernel** implements most of the OS in the kernel, and everything is a giant glob of sharing.
2. The **microkernel** implements different features as client-servers. They enforce modularity by putting subsystems in user programs.

While the microkernel model is easy to debug and may fail in parts only, most systems use the monolithic kernel because of the following reasons:

1. it doesn't matter usually whether the kernel breaks a little or a lot; broken is broken. (how useful is your computer without the file system?)
2. many services require sharing by nature, so it's harder to allocate resources.
3. performance is reduced for client-server overhead, since there are *high* communication costs.
4. once the kernel works, it doesn't matter that you can debug it in modules.
5. everyone already uses monolithic kernels, so no one's willing to put time and effort into testing a new, unproven concept.

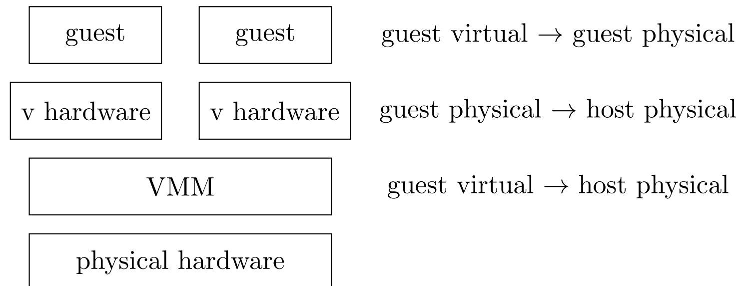
13.2 Virtual machines

How do we deal with bugs in the Linux kernel without redesigning Linux from scratch? We can try to save the machine as a whole, and testing things with virtual machines! Thus, there can be one machine to many kernels.

There is a **host OS** and a **guest OS**. Katrina proceeds to demo that we can crash an Ubuntu VM without crashing the outer OS X.

"Oh my god did you just dab in the middle of class?"—lacurts

Now, let us change our point of view. We can have multiple operating systems running in parallel. There is a virtual machine monitor (VMM) which deals with privileged instructions, allocates resources, and dispatches events.



The guest OS runs in user mode. Privileged instructions throw exceptions, and VMM will trap and emulate. In modern hardware, the physical hardware knows of both page tables, and it directly translates from guest virtual to host physical (it's smart!)

However, there are still some cases in which we cannot trap exceptions. There are several solutions. **Para-virtualization** is where the guest OS changes a bit, which defeats the purpose of a VM. We want to run the actual OS we're emulating. **Binary translation** is also a method (VMWare used to use this), but it is slightly messy. Finally, **hardware support** for virtualization means that hardware has VMM capabilities built-in. The guest OS can directly manipulate page tables, etc. Most VMMs today have hardware support.

> **Question 13.1.** Isn't hardware harder to change? This is a tradeoff. Hardware is fast, and software is slow. Running multiple VMs is very common now, so it makes sense to implement things fast and specialized.

Summary: we can now run many operating systems in parallel. In practical usage, we can use this idea in large data centers, or move virtual machines between computers.

14 February 28, 2017 (R)

14.1 Eraser discussion

We begin with an example.

Thread 1	Thread 2
1 lock(A), lock(B)	1 lock(A), lock(B)
2 v1 = v1 + 1	2 v1 = v1 + 2
3 unlock(A)	3 unlock(B)
4 v2 = v1 + 5	4 v2 = v1 + 6
5 unlock(B)	5 unlock(A)

Eraser will throw an error in this case because the intersection of locks becomes the empty set. Now we consider the extended version of Eraser. There could be false positives and false negatives.

False negative	False positive
• delayed initialization	• private locks
• unexecuted code	• memory reuse
	• benign races

For example, consider the following code.

```
1 if A < 5
2     lock(A)
3     if A < 5
4         A ← A + 1
5     unlock(A)
```

Eraser would flag this, but we see that it's actually fine. So is Eraser actually useful? They tested on several systems, mostly toy things and production servers, but they found that people were generally good about locks. They found lots of false positives.

15 March 1, 2017 (L)

15.1 Performance

Systems are generally designed to meet specific performance goals. Often, a **bottleneck** arises, when one stage takes longer than any others. Bottlenecks often arise from several reasons. First, different components have different growth rates.

Example 15.1 (Chip design)

Often, we can make a chip faster, by shrinking it, but then there is less surface area for heat dissipation. This is compounded by the fact that faster chips dissipate even more heat.

Second, multiple clients may share resources, so there is overhead of providing generality. This leads us to the idea that *when in doubt, we should use brute force.* After all, processing power will undoubtedly increase, so it's better to use an algorithm that is easy to understand.

There are several metrics by which we measure performance.

1. **Utilization** is how much of a resource we're currently taking advantage of. This depends on context, since for the OS, there may be not much "overhead," but to a software, the OS itself is overhead.
2. **Latency** is the time between a change in input propagates to a change in output.
3. **Throughput** is the rate of useful work, or $\frac{1}{\text{latency}}$.

To improve performance, we go through the following steps.

1. Determine whether performance is an issue.
2. Find the bottleneck.
3. Find the next bottleneck, to see if fixing the current one will actually help.
4. If so, fix it. Otherwise, consider redesigning.

There are also some tricks. We could add fast paths for commonly used resources, process requests concurrently (effectively hiding latency), or queue requests and overload resources. If we know the exact use case, we may be able to design for just enough concurrency, but we often don't know, so we have to expect overload and under-utilization.

We can also **batch** requests, to reduce the overhead and take advantage of resource sharing. This goes hand-in-hand with **dallying**, in which we wait for requests to batch up. Finally, we can **speculate** what will happen, a and potentially pre-process outputs. For example, we can speculate that there will be a batch coming soon, and dally until it comes.

15.2 Layering memory

In designing large systems, we often must make tradeoffs between speed, cost, and size. Thus, we design memory in layers, characterized by **capacity** (total bits or bytes), **average random latency** (time to access random memory), **cost** (money per storage), **cell size** (amount retrieved per block read). For example, CPU registers are the smallest, fastest, and most expensive, while remote cold storage is the largest, slowest, and least expensive.

Caches are very similar to virtual memory in concept. Features that help caches be fast are **temporal locality** (access same resource in succession) and **spatial locality** (access similar resources). These are useful since we read in chunks.

The references required to run a program in a given time period are known collectively as its **working set**. If the working set is larger than the primary device, then **thrashing** occurs, where we have to continuously exchange between top and second cache levels. This is inefficient, so we like to avoid thrashing.

Each level of memory is defined by the references to the level, its bring-in and removal policies, and the capacity. A common removal policy is LRU.

Example 15.2 (Disk arm)

Nowadays, another bottleneck is the physical disk arm, moving from memory location to memory location. There are two common methods of optimizing this. First, the arm could just service requests in succession. However, this would leave far-away requests sad, so there's also the elevator method, in which the arm moves one way and then another.

16 March 2, 2017 (R)

We begin the class with some Jeff Dean facts.

> "Compilers don't warn Jeff Dean. Jeff Dean warns compilers."
>
> "Jeff Dean writes directly in binary. He then writes the source code as documentation for other developers."

16.1 Map-Reduce discussion

Map reduce is a model in which we are provided a *map* function and *reduce* function, in the spirit of functional programming.

$$m(k_1, v_1) \to \texttt{list}(k_2, v_2)$$
$$r(k_2, \texttt{list}(v_2)) \to \texttt{list}(v_2)$$

We demonstrate Map-Reduce for counting words.

Example 16.1 (Word count)

Suppose we want to count words. Our key could be "seuss1.txt" and the value could be "the cat in the hat."

```
MAP(K,V)
1  for w ∈ V
2      emit(w, "1")
```

Now our reduce could take inputs "the" and "['1','1']"

```
REDUCE(K,V)
1  result = 0
2  for v ∈ V
3      result t = int(v)
4      (result)
```

> **Question 16.2.** Do reduce operations have to run strictly after all map tasks? No, but we can't finish all reduce tasks until we have all the intermediate pairs.

In terms of practical usage, MapReduce was the "original" version of open-source Hadoop.

17 March 6, 2017 (L)

17.1 Introduction to networking

> "Important concept number one: has anyone played the new Zelda game? It's so good!"—lacurts

There are many, many machines on the Internet, but first we will focus on the network between those machines. In particular, we will focus on the Internet and various protocols.

A **network** is a graph. There are **endpoints**, like our laptops, and **switches**, which deal with many incoming and outgoing connections. We refer to these collectively as nodes.

1. Nodes can **name** each other. A network needs to figure out how to convey location information to nodes, with names and addresses.
2. **Routing** is how each node determines the minimum cost route to every other reachable node. A **routing protocol** determines how this works.[9]

For example, we could find shortest paths using Dijkstra's algorithm.

> "If you don't know Dijkstra's algorithm, you should learn it. It's like, a life skill."—lacurts

Linked-state routing is where we share every single routing table, but it's very inefficient and it has lots of overhead. However, there is the added complexity that each node only knows its immediate neighbors. In addition, the network is constantly changing. Therefore, we require a dynamic, distributed algorithm.[10]

A **packet** is some data and a header. Headers can include lots of information, but source and destination are particularly important. If more packets arrive at a switch than can be handled, switches have queues. And if the queue is full, the package is dropped.

This motivates the concept of a **transport**, which ensures that we can share the network efficiently, fairly, and reliably. We send an **ack** for getting a package and receive an ack for getting the ack. While nothing is guaranteed, this is a best-effort solution.

17.2 History of the Internet

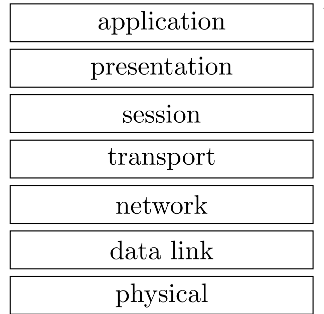
The Internet is the network we'll study. In addition to all these challenges, there are even more!

Early on in the 60s, we needed a survivable communication system, and so there birthed the ARPANET (Advanced Research Projects Agency Network). During the 70s, the ARPANET started to grow... and grow...

In 1978, the desire for flexibility led to **layering**. The most useful layers are physical hardware, the network, transport, and the top-layer application, though there are a total of 7. These are abstractions, which help modularity.

[9] We want to find a minimum spanning tree on the network!

[10] To be pedantic, the *route* is where the next-hop neighbor is, whereas the *path* is the entire path.



In the early 80s, the network grew and grew, which required changes. Flooding with the entire network table was growing infeasible. Then in the mid 80s, late 90s, there was **congestion collapse**, which required congestion control ("hey, chill, stop sending"). The TCP protocol evolved from this era.

In the early 90s, the Internet was beginning to be commercialized. There was a backbone supported by NSF, which didn't want commercial traffic. The BGP protocol came from here, where we could control what types of traffic came across (simply don't tell people about routes). Here, money begins to be a point of contention.

> "Be proud. MIT has control over every IP starting with 18. You could have more machines than IPs, or you could have 16 million IPs like us. We don't have 16 million machines. But we hold onto them because they're a source of pride!"—lacurts

We use a protocol called CIDR to divide IPs in different ways. Things were easy to change back then. All switches were made by Cisco, and most switches were done in software. Today, most switches are done in hardware, since it's very fast, and companies other than Cisco make routers.

In 1993, the Internet was commercialized, so we watch Katrina's favorite video!

There are several contemporary issues we discuss. First, **distributed denial of service** attacks are gamers' worst annoyance.

> "I'm going to tell you my favorite way to mount a DDOS attack. Do not go and mount a DDOS attack."—lacurts

1. Lie about your name. Say you're Karen, so Jack will respond to Karen instead of you.
2. Jack sends a large response to Karen.
3. Run a bot-net that sends many responses to Karen.

Second, **security**. So many things on the Internet are insecure! DNS is not secure. BGP is not secure. Third, **mobility**. When we walk around, our IP addresses change.

> "It's like Startrek!"—lacurts.

18 March 7, 2017 (R)

18.1 DARPA Internet discussion

We review the history of the Internet. The original Internet had several applications in mind: file transfer, remote jobs and login, and voice transfer.

The main design goals were the following.

1. inter-networking
2. fault tolerance / survivability
3. multiple service types
4. heterogenous networks
5. distributed management
6. cost efficiency
7. low-cost host attachment (end-points)
8. accountability

Some aspects they didn't consider include security, performance, and flexibility/evolvability.

There is also a "hourglass" or "narrow waist" idea that relatively unified transport/network layers provide. There are many types of providers on the link layer: satellite, Ethernet, radio, wireless, and coaxial cables. There are many applications, including HTTP, SSH, FTP, DNS, BGP, NTP, AND Telnet. However, there are generally only TCP an UDP for transport; and IP and ICMP for network.

19 March 8, 2017 (L)

19.1 Routing protocols

The goal of a **routing protocol** is to allow every switch to know a minimum cost route to each node, within its routing table.

1. Nodes learn about their neighbors when they send HELLO messages.
2. Nodes learn about other reachable locations via advertisements.
3. Nodes can learn min-cost routes.

19.2 Autonomous systems

A simple of the Internet would consist of many computers, connected to the collective cloud of an Internet that opening shares information. Network attachment points are named using IP addresses, which are topological; **address prefixes** like 18.* can be used to refer to all addresses in some range. Users would then connect to those points, in a friendly, connected graph.

In reality, commercialization renders this untrue and overly optimistic. Instead, there are many **autonomous systems**, each managing its own customers and routing protocols, and interacting at the boundaries through the **Border Gateway Protocol**, or BGP. There are two main ways in which traffic flows through or is shared between AS's.

1. **Transit** is where one AS buys access from a larger AS.
2. **Peering** is where two AS's agree to share routing tables at no cost.

While peering is often mutually beneficial, it requires a tier 1 (global) AS at the root, who can control all traffic in its tree. In addition, while it can save money, it also generates no revenue, and depending on traffic flow ratios, terms may need to be often renegotiated.

ISPs charge customers for access to their routing tables, so there are several types of routes, preferred and not.

1. ISPs earn money off their customers, so **transit customer routes** are always advertised. This means that ISPs will convince the customer that the Internet is at their disposal.
2. ISPs earn no money off leaking a customer free access, just because it needed to forward along a request for someone else. Therefore, **transit provider routes** are not advertised to everyone; just the required.
3. ISPs charge for their routes, so they don't advertise everything in **peer routes**.

This selective transit is made possible by route filters, and leads to the following ranking.

$$\text{costumer} > \text{peer} > \text{provider}$$

19.3 BGP

To control selective sharing, BGP was developed in the days of NSFNET. It had three main design goals.

1. The system must be scalable. Routers should handle any valid IP, and BGP must find DAG paths within a reasonable time.
2. Each AS must be able to enforce and design its own routing policies.
3. AS's should be able to make local decisions, which do not interfere with the network as a whole.

The protocol has several possible messages.

- OPEN is sent after a TCP connection is established. Routers exchange filtered tables.
- UPDATE messages can remove outdated entries or inform about changes.
- KEEPALIVE messages ping for "are you alive."

20 March 9, 2017 (R)

20.1 Resilient overlay networks

An **overlay network** is a network built on top of another existing one, where the virtual links map to actual links in the underlying.

Example 20.1 (Dial-up)
The Internet was an overlay on top of phone lines,

Example 20.2 (Virtual private network)
VPNs emulate being on another network, while still using the local network.

We might use overlay network to achieve a different set of goals, without modifying the underlying structure or protocol. We can also collect data about the network. It's counterintuitive, but we can do *better* than the underlying network.

Why is RON useful in particular? No one has a global topology, since AS's only interface locally.

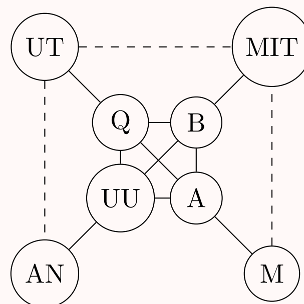
We warm up from some true/false questions.

1. To improve scalability, each RON node only probes other RON nodes that are geographically close. **False**, all nodes probe all others.
2. It's possible on the Internet for the IP path from host A to B to have longer latency than the composition of IP paths from A to C and C to B. **True**.
3. RON's active probing reacts to congestion along paths and reroutes packets around paths with excessive congestion whereas BGP usually does not. **True**.

20.1.1 Failure detection and correction

RON usually just requires 1 node to reroute a failure. RON doesn't scale very well, but we need enough nodes for path diversity.

Example 20.3
We talk about an example.



If the connection between UU and A breaks, BGP will generally detect the error eventually, but it will take a long time. BGP has to scale, so it doesn't want to keep advertising fluctuating routes, so information could be stale.

> **Question 20.4.** Where does the arbitrary 20 seconds come from? Well, closest round number?

20.1.2 Application integration

There are applications that prioritize latency or throughput.

Latency	Throughput
• games • voice • communication • GPS mapping • trading	• media streaming • file transfer • content delivery • peer to peer

We can tailor route tables to prioritize latency or throughput.

20.1.3 Expressive policy routing

Prevent certain applications from using certain links. Packets get classified and tagged.

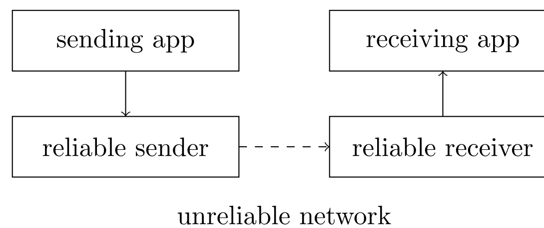
21 March 13, 2017 (L)

21.1 Reliable transport

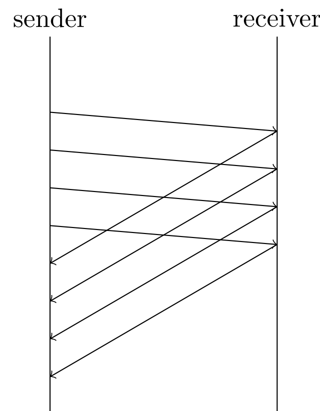
The Internet has lots of problems!

how do we **route** scalably, given policy and economy?	→	BGP
how do we **transport** data, given varying application demands	→	TCP
how do we **adapt** new technologies?	→	next time

We want to receive *one* copy of a byte stream, in order.



Imagine a sender and receiver. We have a window size w of outstanding packets.



There are several bad things that could happen.

- We receive acknowledgements that they received acknowledgements, up to and including packet k. Therefore, if we sent 7,8,9, and 7 gets dropped, then they will keep sending acknowledgements for 6, so we resend 7.
- If an acknowledgement gets lost, say acknowledge packet 8, then it's fine! Except the sender doesn't know, so it will resent anyways.
- If something gets super delayed, it's essentially lost, so there are duplicates, but the receiver just suppresses them.

21.2 Congestion control

Reliable transport is easy! But what happens when packets get dropped? If we send too few, the network is underutilized. If we send too many, the network becomes congested. We can poke the network, but algorithms are hard.

> "You shouldn't just accept that I named you Glen!"—lacurts

The issue is: how can a single reliable sender, using a sliding window protocol, set its window size to be not too big, not too small? The solution was called **congestion control**, which is to control the source rate, to avoid congestion. We want to avoid packet drops.

> "This is so going to be on the midterm. Oh look how many heads just looked up!"—lacurts

There are two objectives of congestion control.

1. we want to use the network efficiently: minimize packet drops and delay, while maximizing utilization.
2. we want to distribute network resources fairly.

Today, we introduce **end-to-end** congestion control. Switches are dumb, but senders are smart. Senders will increase their window size until they experience congestion; then they will back off and try again. If packets are being dropped, then queues are *probably* full. Every round-trip time if packets are dropped, we increase window size by one. Otherwise, we decrease by half. TCP uses the **additive-increase, multiplicative-decrease** rule.

There are some slight issues. Nodes are different distances away, so increases happen in different rates. We take a long time to ramp up. At the beginning of a connection, we use **slow start**, which is a slight misnomer since we exponentially increase how much we sent.[11]

[11] In the past, they sent *everything* at the start, so this is indeed slow compared to that.

22 March 16, 2017 (R)

22.1 Bufferbloat discussion

Buffers are essential for absorbing bursty Internet traffic, but they can lead to a condition known as **bufferbloat**. When a buffer is full, droptail methods increase the time for packet-drop messages to be sent, resulting in many dropped packets and delayed response to full queues. In effect, buffers become part of the pipe. Active queue management (AQM) methods exist, but they are not widely deployed, and requirements change too quickly.

Bufferbloat is exacerbate by the large buffer sizes throughout the Internet. Memory is inexpensive, but it leads to even longer delays. While long flows are not severely affected, latency-sensitive flows become sad. We analyzed two contrasting papers by Getty and Allman.

Getty

1. Bufferbloat can happen, and it's a huge problem.
2. Paper comes from personal communication with experts and spot-checking author's own residential network and community networks.
3. Provides many worked examples, with theoretical analyses and calculations.
4. Proposes a solution, but portends future doom.

Allman

1. Bufferbloat is a worst-case scenario; it will never be such a big issue in the amortized average case.
2. Analyzed large CCZ network and ran experiments.
3. Bufferbloat more prevalent in residential networks, but not of primary concern.
4. LIttle effect of change in initial window size on bufferbloat. RTT fluctuation shows that TCP is working.

23 March 20, 2017 (L)

23.1 File sharing

In the simple case, a client sends a request for a file, and the server sends a response, which is the file. However, there are some drawbacks and amendments.

- The server is a single point of failure. We could buy more servers and direct clients to different servers. This is known as a **content distribution network**.
- The client-server model is not very scalable. Instead, we could have a **peer-to-peer** network. For example, BitTorrent.

23.2 BitTorrent

How do we keep track of torrenting files? There are trackers. Some peers are special, called **seeders**, who have the entire file already. The file is broken down into numerous blocks. Typical block sizes are around 16 kb, which is the size of the unit of exchange. Peers exchange bitmaps about which blocks they have. You don't need to download the blocks in order.

People might not want to share files after they download them; they don't want to waste their bandwidth. Therefore, there are incentives: someone can only download a block if they also upload a block. In round t, we receive n blocks. In the next round, we upload to k peers who give us the most.

We need to bootstrap new peers somehow. Every peer reserves some of their bandwidth to give out for free. In BitTorrent, this process is called **unchoking**.

- If peers fail, that's fine. There are many others.
- If a tracker fails, then no one else can join the swarm. This is a central point of failure!

We could have a distributed hash table instead.

23.3 VoIP

On the Internet, there are many **network address translators** (NAT). Each NAT has some public IP address.

I'm really tired so I give up on taking notes today sorry.

24 March 21, 2017 (R)

24.1 Data center TCP discussion

Data centers present an interesting "special case" for transporting data. They are characterized by the following.

- Low RTTs (delay is not a primary issue).
- Anticipated traffic patterns, with large traffic flows (throughput) and short, latency-sensitive flows (worker updates, configuration).
- Low statistical multiplexing: there are relatively few paths active at any given time.

There are several issues that may arise in such an environment. These include incase, queue buildup, and buffer pressure. However, DCTCP is able to resolve them using its adjustable window size. We compare different versions of TCP and AQM protocols.

	notify	react
droptail	drop packet	halve window size
RED	drop packet	randomly drop packets above threshold
ECN	mark packet	cuts window size
DCTCP	mark packet	cuts window size based on fraction marked

25 March 23, 2017 (R)

25.1 Content delivery networks discussion (Akamai)

We read Akamai's original paper, detailing its infrastructure and system design. This was written in 2010, when streaming was just beginning to dominate the Internet, over P2P traffic. If we `ping` some websites like `whitehouse.gov`, we see that traffic is actually routed through several Akamai servers. Nowadays, Akamai routes about a quarter of the Internet.

Rewinding, traditional Internet transport relies on TCP and BGP, which don't respond well to change, poorly handle bursty traffic, and are unaware of the underlying network topology. The Internet is made worse by the unprofitably of the "middle mile," between ISPs and end-user networks. These issues are exacerbated by modern applications, which require speed, reliability/consistency, management, and scalability. Such applications include:

1. video streaming
2. web commerce
3. interactive and/or personalized applications
4. collaborative software
5. static content

There are several methods one could distribute content.

1. Centralized content server, with possible mirrors. Load balancing is a huge issue.
2. Distributed content servers. These may be *okay* depending on the use case.
3. Very distributed content delivery network. Akamai chooses this model.
4. Peer to peer. Some applications such as streaming are hard, though there have been more recent developments in that.

Akamai, as a content delivery network, helps improve performance by focusing on edge servers, close to the clients. They have built a very large, distributed system. An interesting tidbit is that Netflix used to use Akamai, but it became more cost-efficient and effective to eventually build their own content delivery network.

26 April 3, 2017 (L)

26.1 Fault tolerance

We will develop a systematic way to tolerate faults, so that our system keeps running despite failures. In some cases, we also want to recover from failures. Here, we consider normal failures, which do not include adversarial attacks.

1. **Identify all possible faults.** A computer could crash. Data centers could get wiped out of tsunamis. There are a lot of them.
2. **Detect and contain the faults.** It would be problematic if the data got mangled and we never knew.
3. **Handle the fault.** Decide on a plan. Fix it. There are several options.
   (a) Do nothing.
   (b) Fail fast.
   (c) Stop.
   (d) Mask the error.

There are several disappointing things.

1. We can never guarantee that any component is perfect; our components are *always* unreliable. Therefore, we only have probabilistic guarantees.
2. In addition, reliability comes at a cost. More reliability costs more. The most common tradeoff is complexity.
3. This process is hard, since we simply cannot find all the faults. This process will be iterative.
4. Fault tolerance still requires that *some* code is correct on a low level. LaTeX can crash; it's sad but okay. Storing files should be safe though.

We can quantify reliability through several metrics. Given MTTF (mean time to fail) and MTTR (mean time to repair),

$$\text{availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}.$$

There are several methods for fault tolerance, one of which is redundancy. Extra bits help error detection and correction. The Internet is redundant; there are multiple paths between almost every source and destination.

The cost of disk failure is very high—we can't get the data back! But it doesn't happen *too* frequently. Generally, companies report MTTF, but that is calculated with the assumption that disks are equally likely to fail no matter when. Disks are most likely to fail when they're first created (**infant mortality**) and after around 5 years (**burnout**).

> "We're commited to failures. We're going for big failures here"—lacurts

We want redundancy? Just buy another disk, ya? Read from one of them. What if it fails? We assume we can detect failure. Now we buy a new disk and copy everything. What are the tradeoffs? Writes are a bit slower, the maximum of the two. We bought a whole new disk for no more new data! This method is part of RAID (**redundant array of independent disks**).

Our example is known as **RAID-1**, where we mirror a single disk. This requires $2n$ disks to backup n's worth.

"Your whole life is just 0s and 1s"—lacurts

We can do better. We could buy a single new disk, the **parity disk**, where we XOR all sectors. This is known as **RAID-4**. If a single disk fails, we just XOR all the rest. If the parity disk fails, we can recover it in the same way. We only need to buy a single new disk! The downside is that all writes hit the parity disk, which means that writing in parallel is harder. This idea is useful pedagogically.

We spread out parity sectors across disks in **RAID-5**. There are performance benefits, as writes are spread across disks.

What if the network in between fails? Consistency may be an issue.

27 April 10, 2017 (L)

27.1 Atomicity

Our goal is to build reliable systems from unreliable components. We would like to design **transactions** that provide atomicity and isolation, while not hindering performance.

atomicity	→	shadow copies (simple, poor performance), meh but some people use
isolation	→	today's topic

In addition, these transaction-based systems must be distributed to run across many machines.

We keep a **log** so that aborted operations can be reverted, and their effects erased. There are records: update and commit. Update records have the old and new values. Commit records indicate that a transaction has been commited. There could also be an abort record.

- On begin, allocate a new ID.
- On write, append new entry to log.
- On read, scan backwards, find last thing commit value.
- On commit, write a commit record.

At the moment, we don't have to clean up anything for recovery. Appending to logs is fast, so write performance is pretty good. Read performance is abysmal. Recovery performance is good.

> "The orange chalk will the most exciting part of your day in 6.033 today."—lacurts

27.2 Write-ahead logging

We use cell storage. Note that this is not a cache. This is extra storage on disk. We **log** an update when it's written to log, and we **install** an update when it's written to cell storage. Logging appends while installing overwrites. Therefore, we *must* log before we install. This is known as the **write-ahead logging** (WAL) protocol. This is *not* atomic! So recovery involves finding the errors (losers) and fixing them. If a machine keeps crashing and recovering, it's fine! It's idempotent.

Read performance improves and is good. Writes are slightly worse. Recovery got a lot slower. We want to improve write and recovery a little, so we add a cache.

We write to cell storage, write to log, and read from cache. Performance-wise, reads are perhaps even better. Writes are about the same. Recovery is even worse.

So we make one last update: write checkpoints and truncate the log.

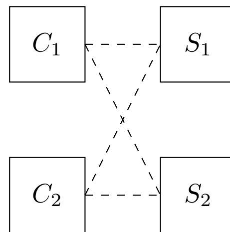
TODO diagrams???

28 April 24, 2017 (L)

28.1 Availability through replication

atomicity	$\rightarrow$	shadow copies, logs
isolation	$\rightarrow$	two-phase locking

We want to increase the availability of our systems, and the solution is replication. Today, we introduce **single-copy consistency**, which means that code will execute as if there were only a single copy. Internally, they may be discrepancies, but those are masked to the client.

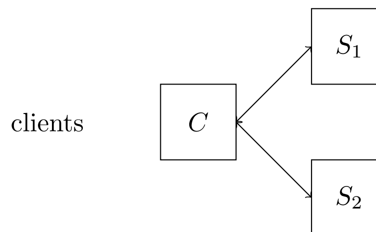


A major problem is that replica servers can become inconsistent. Thus replicate state machines have several goals.

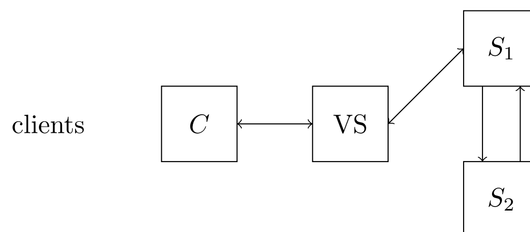
1. Each server starts with the same initial state S_1.
2. All operations are deterministic.

We assume that replicas fail independently, which is not realistic, but it simplifies master.

There exists a coordinator with a log. All clients send their requests through that coordinator, which communicates with the replicas and responds to clients.



Imagine a network failure that leads to a network partition. This is troublesome! So we could have a **view server**, which determines which replica is the primary.



It has a view table, which tells servers their role. Copies will compute, send copies, and wait for acks before replying to the coordinator.

We try out some test cases.

1. The primary fails. The view server switches to backup.
2. A network partition cuts off the primary entirely, it's as if the primary failed.
3. A network partition causes $S_1 \to S_2$ to remain fine, but $S_2 \to S_1$ is broken. The primary must wait for the backup, but any replica must reject coordinator requests. We can change the primary server.
   (a) What if S_2 is the designated new primary, but S_1 still thinks it's the primary? S_2 won't accept the updates from S_1.
   (b) S_2 ignores.

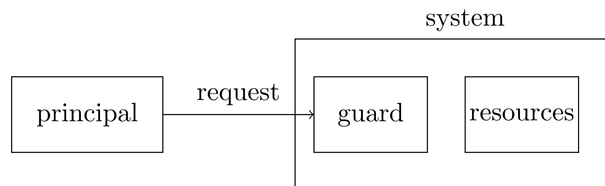
Why do we use a view server instead of a fancier coordinator? We want to replicate coordinators, so we need a single view server. But what if the view server fails? It's a bit crazy to expect a single machine to do all this work.

Curious to learn more? Go to recitation tomorrow!

29 May 1, 2017

29.1 Principal authentication

So far, we've learned that security is hard! There is the **guard model**, in which the guard typically provides **authentication** and **authorization**. Today, we will discuss authentication, or *how does the system verify identity*?



We use passwords for authentication. Consider an adversary who just attacks our server, not the network. How do we store passwords? Naively, we could just store a direct mapping table.

username	password
dianez	frand
vrm	otherfrand
rmwu	me

> "If you build a system that stores plain text passwords, I will not claim you as a 6.033 student."—lacurts

Instead, we consider hashing. A hash function h has the following properties.

1. h is deterministic.
2. Given x, we can compute $h(x)$, preferably fast.
3. Given $h(x)$, we cannot determine x (not computationally tractable). That is, h is one-way.
4. If x_1, x_2 are not equal, then neither are their hashes with high probability. That is, h is collision-resistant.

username	md5 hash (slow)
dianez	693688ccaefd99960eae1b6668b9cd58
vrm	ca3b9ca07f8f6fb4d700e7c0f5d2adfe
rmwu	ab86a1e1ef70dff97959067b723c5c24

However, these hashes are quite slow. Also, users are the worst! They don't select passwords uniformly at random. Instead, we can salt the hashes. Whenever users create an account, we create a salt for them. We then hash the password, concatenated with the salt. This is how passwords are stored!

username	salt	md5 hash (slow)
dianez	MRX0R93QS43T	1ae61e044f9373c9860e1d420d150ce2
vrm	2FX3H8JCIOAW	ab402ecd620c536048d1efae8bb5abb1
rmwu	O7E6I6BN7OBV	f6b3247ee4e68d20aca0228b1e80012a

Now how does authentication actually work? We don't enter our passwords all the time because it's not secure to keep typing and transferring it, and people would choose trivial passwords. Therefore, we like cookies or authentication tokens. We assume that our server is safe, and it uses a **challenge-response** protocol.

1. The server knows our password. It asks us to compute a hash with a random number.
2. We compute the hash and send it over.
3. The server verifies.

Now assume that our server is not valid. Even if the server knows our hash, it cannot compute our password.

Finally, how are accounts created? MIT makes it hard to get an account: you need to be admitted first. But most websites just ask for an email.

> "My security answer questions are random strings!"—lacurts

30 May 15, 2017

30.1 Ransomware

There's an evil creature lurking around called ransomware. It locks up your computer and demands payment in bitcoin. A researcher found that the domain name for this was unregistered, so he registered it, and everything stopped working! The bots were smart and would stop if someone registered the domain name, so they could protect themselves.

> "Last time I checked, they raised—no not raised, extorted $33 thousand."—lacurts

It's hard when the computer locked is some hospital database.

30.2 Anonymity

Today we focus on the Tor network. There are two ways to encrypt data.

1. Symmetric key cryptography (encrypt with k, decrypt with k). Both parties must keep the key a secret!
2. Public key cryptography (RSA)

Tor's goal is to assure anonymity over an insecure channel. Suppose Rachel wants to surprise Tony.

1. No packet should say "from Rachel, to Tony."
2. No entity in the network receives a packet from Rachel and sends it directly to Tony.
3. No entity in the network should keep state that links Rachel to Tony.
4. Data should not appear the same across multiple packets.

First we can use a proxy. Rachel sends to P and P sends to Tony.

$$\text{Rachel} \longrightarrow P \longrightarrow \text{Tony}$$

However, an adversary could quickly link events up. We could send to a network of proxies.

$$\text{Rachel} \longrightarrow P_1 \to P_2 \to P_3 \to \text{Tony}$$

We also assume that packets have enough state for Tony to send Rachel a reply. An adversary sees lots of network traffic, but it's harder to tell who sent what. But packets are still the same! Packets look the same, so it's easy to track them from proxy to proxy.

So now we add layers of encryption to each packet and proxy.

1. Rachel wraps the gift in Tony, P_3, P_2, and P_1's public keys, in that order.

2. P_1 unwraps the gift, still doesn't know what the gift is.
3. Then P_2 and P_3 unwrap theirs.
4. It finally gets to Tony, who unwraps the last layer himself.

Each proxy strips off a layer of encryption and edits the header. This is known as "onion routing," and Tor is known as the onion router! omg

Tor says that any breach is as bad as no trust at all. So all users use a fixed number of entry guards. It's good to tell users exactly what they don't defend against.