Introduction
Principles of System Design

COS 518: *Advanced Computer Systems*
Lecture 1

Mike Freedman

Goals of this course

- Introduction to
  - Computer systems **principles**
  - Computer systems **research**
    - Historical and cutting-edge research
    - How “systems people” think
- Learn how to
  - **Read** and **evaluate** papers
  - **Give talks** and evaluate talks
  - **Build and evaluate** systems

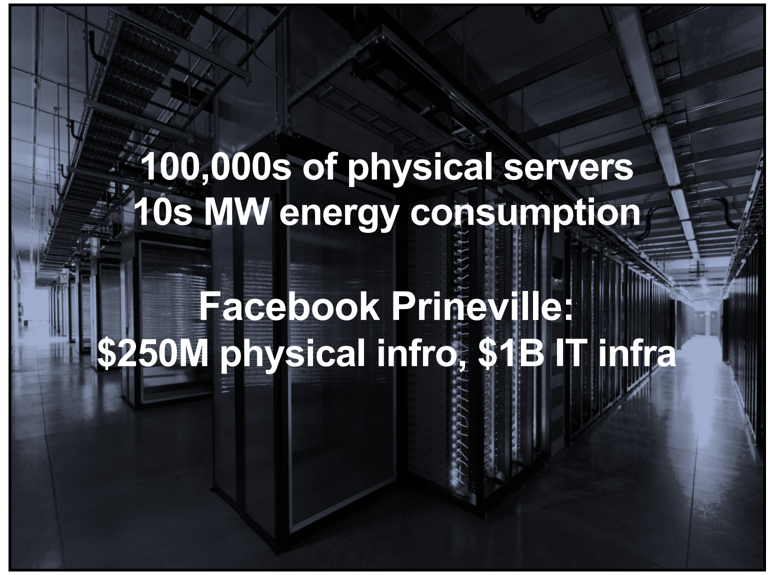
2

What is a system?

- **System**
  - Inside v. outside: defines interface with environment
  - A system achieves specific external behavior
  - A system has many components
- This class is about the design of **computer** systems
- Much of class will operate at the design level
  - Guarantees (semantics) exposed by components
  - Relationships of components
  - Internals of components that help structure

Backrub (Google) 1997

4



The central problem: Complexity

- Complexity's hard to define, but symptoms include:

1. Large number of **components**
2. Large number of **connections**
3. Irregular **structure**
4. No short description
5. Many people required to design or maintain

Course Organization

8

Learning the material

- Instructors
  - Professor Mike Freedman
  - TA Daniel Suo
  - Office hours immediately after lecture or by appt
- Main Q&A forum: www.piazza.com
- Optional textbooks
  - *Principles of Computer System Design*. Saltzer & Kaashoek
  - *Distributed Systems: Principles and Paradigms.* Tanenbaum & Van Steen
  - *Guide to Reliable Distributed Systems.* Birman.

9

Format of Course

- Introducing a subject
  - Lecture + occasional 1 background paper
  - Try to present lecture class *before* reading
- Current research results
  - Signup to read 1 of ~3 papers per class
  - Before class: Carefully read selected paper
  - During class: 1 person presents, others add to discussion
  - During class (before presentations): answer a few questions about readings ("quizlet")

10

Course Project: Schedule

- Groups of 2-3 per project (will finalize tonight)
- Project schedule
  - Team selection (2/10, Friday)
  - Project proposal (2/24)
  - Project selection (3/3): Finalize project
  - Project presentation (before 5/16, Dean's Date)
  - Final write-up (5/16, Dean's Date)

11

Course Project: Options

- **Choice #1: Reproducibility**
  - Select paper from class (or paper on related topic)
  - Re-implement and carefully re-evaluate results
  - See detailed proposal instructions on webpage
- **Choice #2: Novelty** (less common)
  - Mus be in area closely related to 418 topics
  - We will take a **narrow** view on what's permissible
- Both approaches need working code, evaluation

12

Course Project: Process

- **Proposal selection process**
  - See website for detailed instructions
  - Requires research and evaluation plan
  - Submit plan via Piazza, get feedback
  - For "novelty" track, important to talk with us early
- **Final report**
  - Public blog-like post on design, eval, results
  - Likely posted to Medium
  - Source code published

13

Grading

- 15% paper presentation(s)
- 15% participation (in-class, Piazza)
- 20% in-class Q&A quizlets
- 50% project
  - 10% proposal
  - 40% final project
    - 20% overall, 10% presentation, 10% write-up

14

Organization of semester

- Introduction / Background
- Storage Systems
- Big Data Systems
- Applications

15

Storage Systems

- Consistency
- Consensus
- Transactions
- Key-Value Stores
- Column Stores
- Flash Disks
- Caching

16

Big Data Systems

- Batch
- Streaming
- Graph
- Machine Learning
- Geo-distributed
- Scheduling

17

Applications

- Publish/Subscribe
- Distributed Hash Tables (DHTs)
- Content Delivery Networks
- Blockchain
- Security
- Privacy

18

Principles of System Design

19

Systems challenges common to many fields

1. **Emergent properties ("surprises")**
   – Properties not evident in **individual** components become clear when **combined** into a system
   – **Millennium bridge,** London example

Millennium bridge

- Small lateral movements of the bridge **causes** synchronized stepping, which **leads to** swaying
- Swaying **leads to** more forceful synchronized stepping, **leading to** more swaying
  - – Positive feedback loop!
- Nicknamed *Wobbly Bridge* after charity walk on Save the Children
- Closed for two years soon after opening for modifications to be made (**damping**)

Systems challenges common to many fields

1. Emergent properties (“surprises”)

2. **Propagation of effects**
   - – **Small/local** disruption → **large/systemic** effects
   - – Automobile design example (S & K)

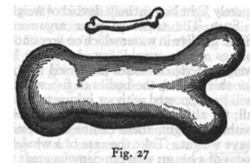
Propagation of effects: Auto design

- **Want a better ride** so increase tire size
- Need larger trunk for larger spare tire space
- Need to move the back seat forward to accommodate larger trunk
- Need to make front seats thinner to accommodate reduced legroom in the back seats
- **Worse ride** than before

Systems challenges common to many fields

1. Emergent properties ("surprises")
2. Propagation of effects
3. **Incommensurate scaling**
   – Design for a smaller model may not scale

Galileo in 1638



"To illustrate briefly, I have sketched a bone whose natural length has been increased three times and whose thickness has been multiplied until, for a correspondingly large animal, it would perform the same function which the small bone performs for its small animal…

Thus a small dog could probably carry on his back two or three dogs of his own size; but I believe that a horse could not carry even one of his own size."

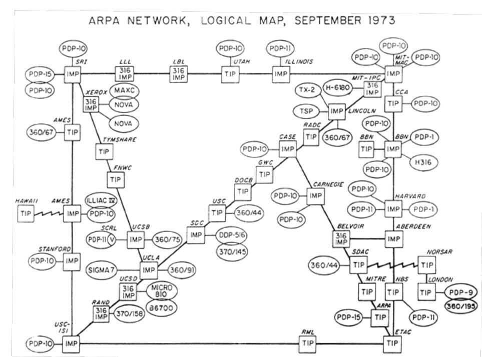
—Dialog Concerning Two New Sciences, 2nd Day

Incommensurate scaling

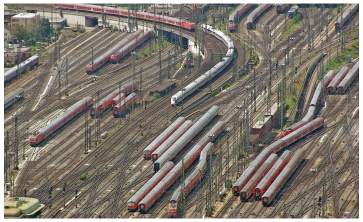
- **Scaling a mouse into an elephant?**
  – Volume grows in proportion to O(x^3) where x is the linear measure
  – Bone strength grows in proportion to cross sectional area, O(x^2)
  – [Haldane, "On being the right size", 1928]
- Real elephant **requires** different skeletal arrangement than the mouse

Incommensurate scaling: Scaling routing in the Internet

- Just **39 hosts** as the **ARPA net** back in **1973**



Incommensurate scaling:
Scaling routing in the Internet

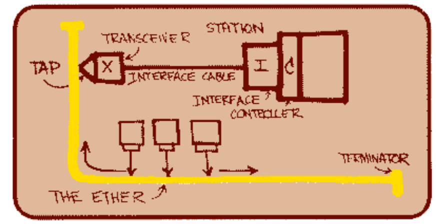


- Total size of routing tables (for shortest paths): **O(n^2)**
- Today's Internet: Techniques to **cope with scale**
  - Hierarchical routing on network numbers
    - 32 bit address =16 bit network # and 16 bit host #
  - Limit # of hosts/network: Network address translation

29

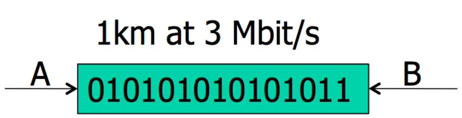
Incommensurate Scaling: Ethernet

- All computers share single cable
- Goal is reliable delivery
- **Listen-while-send** to avoid collisions



Will listen-while-send detect collisions?

- 1 km at 60% speed of light is 5 μs
  - **A** can send 15 bits before first bit arrives at **B**
- Thus **A** must keep sending for 2 × 5 μs
  - To detect collision if **B** sends when first bit arrives
- Thus, min packet size is 2 × 5 μs × 3 Mbit/s = 30 bits



From experimental Ethernet to standard

- Experimental Ethernet design: **3 Mbit/s**
  - Default header is 5 bytes = 40 bits
  - No problem with detecting collisions
- First Ethernet standard: **10 Mbit/s**
  - Must send for 2 × 20 μs = 400 bits
    - But header is just 112 bits
  - **Need for a minimum packet size!**
- **Solution: Pad packets** to at least 50 bytes

Systems challenges common to many fields

1. Emergent properties ("surprises")

2. Propagation of effects

3. Incommensurate scaling

4. **Trade-offs**
   - Many design constraints present as trade-offs
   - Improving one aspect of a system diminishes performance elsewhere

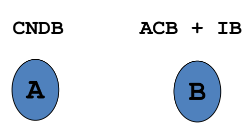
Binary classification trade-off

- Have a ***proxy signal*** that imperfectly captures **real signal of interest**
- **Example:** Household smoke detector

		Real categories: fire	Real categories: no fire
Proxy categories	detector signals	TA: fire extinguished	FA: false alarm
	detector quiet	FR: house burns down	TR: all quiet

Sources of complexity

1. **Cascading and interacting requirements**
   - **Example:** Telephone system
     - Features: Call Forwarding, reverse billing (900 numbers), Call Number Delivery Blocking, Automatic Call Back, Itemized Billing
   - **A** calls **B, B** forwards to 900 number, who pays?



- A calls B, B is busy
- Once B done, B calls A
- A's # appears on B's bill

Interacting Features

- Each feature has a spec
- An interaction is bad if feature X breaks feature Y
- These bad interactions may be fixable…
  - But many interactions to consider: huge complexity
  - Perhaps more than n^2 interactions, *e.g.* triples
  - Cost of **thinking about / fixing interaction** gradually grows to dominate software costs
- Complexity is super-linear

Sources of complexity

1. Cascading and interacting requirements

2. **Maintaining high utilization of a scarce resource**
   - **Ex:** Single-track railroad line through long canyon
     - Use pullout and signal to allow bidirectional op
     - But now need careful scheduling
     - **Emergent property:** Train length < pullout length

Coping with complexity

1. **Modularity**
   - Divide system into ***modules,*** consider each separately
   - Well-defined interfaces give flexibility and isolation

- Example: **bug count** in a large, **N-line** codebase
  - Bug count $\propto$ N
  - Debug time $\propto$ N $\times$ bug count $\propto$ $\mathbf{N^2}$

- Now divide the N-line codebase into **K** modules
  - Debug time $\propto$ $(N / K)^2 \times K = \mathbf{N^2/K}$

Coping with complexity

1. Modularity

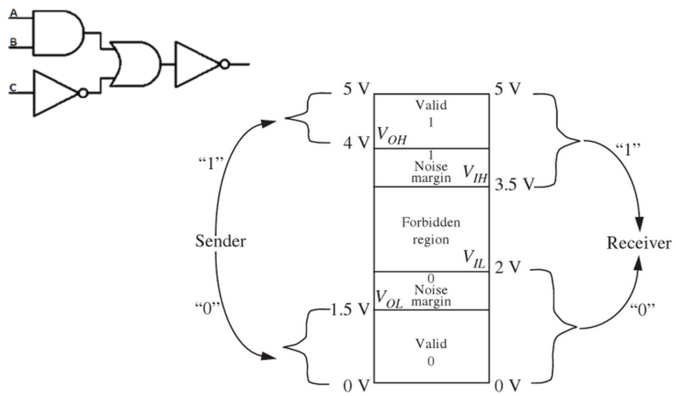
2. **Abstraction**
   - Ability of any module to treat others like "black box"
     - Just based on interface
     - Without regard to internal implementation
   - Symptoms
     - Fewer interactions between modules
     - Less ***propagation of effects*** between modules

Coping with complexity

1. Modularity

2. **Abstraction**
   - **The Robustness Principle:**
     Be tolerant of inputs and strict on outputs

Robustness principle in action: The digital abstraction



Coping with complexity

1. Modularity
2. Abstraction
3. **Hierarchy**
   - Start with small group of modules, assemble
     - Assemble those assemblies, etc.
   - Reduces connections, constraints, interactions

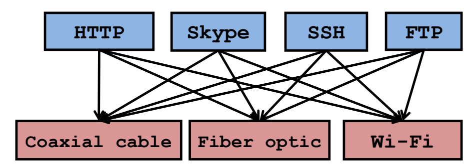
Coping with complexity

1. Modularity
2. Abstraction
3. Hierarchy
4. **Layering**
   - A form of modularity
   - Gradually build up a system, layer by layer
   - **Example: Internet protocol stack**

Layering on the Internet: The problem

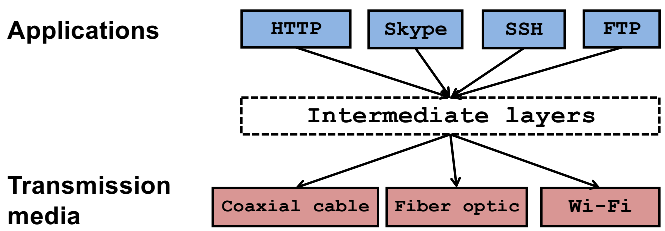
Applications

Transmission media



- Re-implement every app for every new tx media?
- Change apps on any change to tx media (+ vice versa)?
- **No!** But how does the Internet design avoid this?

Layering on the Internet: Intermediate layers provide a solution



- Intermediate layers provide abstractions for app, media
- New apps or media need only implement against intermediate layers' interface

Computer systems: The same, but different

1. **Often unconstrained by physical laws**
   - Computer systems are **mostly digital**
   - **Contrast:** **Analog** systems have **physical limitations** (degrading copies of analog music media)
   - Back to the **digital static discipline**
     - Static discipline **restores signal levels**
     - Can **scale** microprocessors to billions of gates, encounter new, **interesting emergent properties**

46

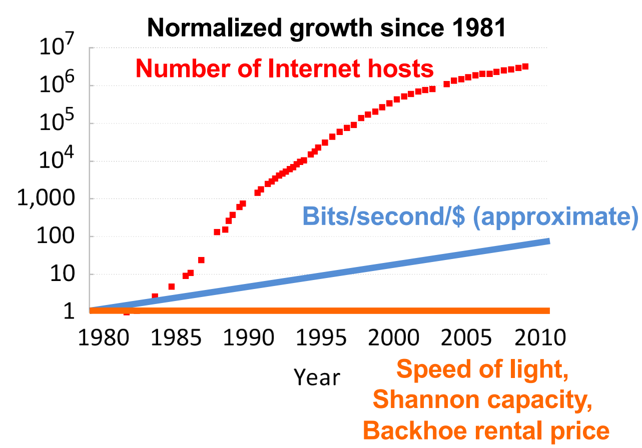
Computer systems: The same, but different

1. Often unconstrained by physical laws

2. **Unprecedented *d*(technology)/*dt***
   - Many examples:
     - Magnetic disk storage price per gigabyte
     - RAM storage price per gigabyte
     - Optical fiber transmission speed
   - **Result:** Incommensurate scaling, with system redesign consequences

47

Incommensurate scaling on the Internet



Summary and lessons

- **Expect surprises** in system design
- There is **no small change** in a system
- 10-100 × increase? ⇒ perhaps re-design
- Complexity is **super-linear** in system size
- Performance cost is super-linear in system size
- Reliability cost is super-linear in system size
- **Technology's high rate of change** induces incommensurate scaling

Paper Readings:

Worse is Better!
Worse is Worse?

50

Setting: The two approaches

MIT approach

- **Simplicity:** Simple in both implementation and **especially interface**
- **Correctness: Absolutely** correct in **all** aspects
- **Completeness:** Cover **all** reasonably expected cases, even to **detriment** of simplicity

New Jersey approach

- **Simplicity:** Simple in both interface and **especially implementation**
- **Correctness:** Correct, but slightly better to be **simple**
- **Completeness:** Cover **as many** cases **as is practical**
  – Sacrifice for **simplicity**

51

Worse is better!

- What does the following compute (**x** is an int): `x + 1`
  – **Scheme:** Always calculates an integer one larger than **x**
  – **Most others** incl **C:** Something like `(x + 1) mod` 2^{32}
- **C: simple** implementation, **complex** interface
  – This is the **key tradeoff** that Gabriel describes
  – Probably not what programmer actually wanted
  – But, **works in the common case**
  – Most languages follow the New Jersey approach!

52

Worse is worse!

- `fgets(char *s, int n, FILE *f)` **versus** `gets(char *s)`
  - `fgets` **limits** length of stored string stored to size <= `n`
  - `gets` stores in `s` **however many** chars from `stdin` are ready to be read

- *Which is the MIT approach vs. New Jersey approach?*
  - `gets` has caused many **buffer overflow security exploits**
  - For security, "the right thing" is the only thing!

53

Hints for Computer System Design

Butler Lampson

54

Systems versus algorithms

- Computer **systems differ from algorithms**
  - External interfaces are less precisely designed, more **complex,** more **likely to change**
  - Much **more internal structure**, interfaces
  - Measure of success much less clear

- And, principles of computer system design are much **more heuristic, less mathematical**

55

Interfaces

- Most of hints depend on notion of ***interface***
  - Separates ***clients*** of an abstraction from the ***implementation*** of that abstraction

- **Interface design** is most important part of system!

- Interfaces should be:
  1. Simple
  2. Complete
  3. Admit sufficiently small and fast implementation

56

Keep it simple

- In other words, follow the New Jersey approach
- Do **one thing at a time,** and do it well
- **Don't generalize:** generalizations are usually wrong, lead to unexpected complexity
- Interface **mustn't promise more** than the implementation knows how to deliver

Continuity

- *As a system changes, how do you manage change?*
- **Keep basic interfaces stable**
- If change interfaces, **keep a place to stand**
  - ***Compatibility package*** (a.k.a. ***shim layer***) implementing old interface atop new interface

Implementation

- **Plan to throw one away (you will anyhow)**
  - Brooks' observation in The Mythical Man-Month
  - Revisit old design decisions with benefit of hindsight
- **Keep secrets** of the implementation
  - **Assumptions** about the implementation that clients are **not allowed** to make (b/c can change)
- Instead of generalizing, **use a good idea again**

Handling all the cases

- Handle **normal** and **worst** cases **separately:**
  - The **normal** case **must be fast;**
  - The **worst** case must **make some progress**

Wednesday:

Everybody reads Saltzer E2E

61