CSE 130

Principles of Computer System Design

Elijah Hantman

Systems, Complexity and Fundamental Abstractions

What is a System?

- Set of interconnected components
- What is a component depends on the problem and scope
- For hardware designers each individual device could be a component
- For CPU designers individual transistors may be component
- For Servers different stages of processing a request may be a component

Complexity

- Large systems with many components can make systems hard to understand
- No unified measure of complexity
- Complexity puts limits on what you can build and how
- Important to understand what is complex and how to mitigate or remove complexity

Common Issues:

- Emergent Properties
  - Properties which arise from the confluence of multiple components
  - Example: Millenium Bridge in London which ended up being extordinarily unstable due to the fact that people walking on the bridge were pressured to walk in sync which generated harmonic motion.
  - Example: Conway's Game of Life, simple rules give rise to a Turing complete system which can compute any problem and simulate all kinds of unique and new patterns
- Propogation of Effects
  - Problems in one component can affect other components
  - Example: Electric companies, by connecting their systems they can load balance and hopefully reduce failures. However it also means that if one system is brought down by a failure all the systems go down
  - Example: Networking, if a network connection fails or a component goes down, it can force changes in other components to deal with the failure to send or recieve data from that particular host
- Incommensurate Scaling
  - Different components in a system scale in different ways as the system scales up.
  - Example: As a database grows the time to query data may increase faster than the size leading to performance issues.
  - Example: As CPUs grew faster, memory grew faster slower, leading to massive differences in computer speed vs. memory bandwidth.
  - Common in many systems especially among heterogenous components since different types of work fundamentally have different costs.
- Trade-Offs
  - Goals may conflict
  - Example: Bigger displays on phones look better but may cost more battery

  - Example: Merge Sort scales well, but a recursive implementation imposes a burden on memory
  - Example: Insertion sort scales poorly, but has very small overhead making it faster for small amounts of data.
  - Example: Spam filters have to balance rejecting spam emails and accepting non spam emails, it can be very bad to reject too many emails but the spam filters are pointless if they don't reject enough emails

Signs of Complexity

- No quantitative measure
- Irregular

Large Number of Components

- Size affects comprehensibility
- It is an indicator but not a direct measure
- Depends largely on how components are connected

Large Number of Interconnections

- Each interconnection places a burden on the person trying to understand the system

Many irregularities

- If there are consistency and patterns in the interconnections and components it can be easy to conceptualize
- The more exceptions the more likely the system is difficult to follow
- Each exception takes up some mental bandwidth

Long Descriptions

- If the best description requires many properties it may indicate complexity
- Kolmogorov Complexity - length of the shortest specification
- May be a reflection of other signs

Large Team of People

- The more people the more likely the system will be complex
- The boudaries between people are reflected in the product, leading to more components and more complexity

Sources of Complexity

- Cascading and interacting list of requirements
  - Each requirement may be minimal
  - The Problem is that each requirement may interact with some number of previous requirements causing exponential growth in the possible interactions
  - Example: Phones
    * corded phones have a single purpose
    * flip phones have basic apps

    * smart phones are full computers and can do hundreds of tasks
  - Must avoid excessive Generality, systems that can do anything are not particularly useful for anything. The more general the more complicated and more difficult it is to create a good tool.
- Maintaining high utilization
  - If we desire high utilization we need more complexity
  - Law of Diminishing Returns: the closer we get to some maximum the more effort it requires
  - Utilization is less percentage and more odds, if we double the amount of complexity and utlization, it halves the idle time. This means that by doubling the complexity we get highly diminishing returns.

How to mitigate Complexity

- Modularity
  - Divide and conquer
  - design the system as a small number of interacting subsystems, reduces the amount of the system under consideration at any one point
  - Reduces the global problem to small set of interactions, allows ignoring global problem when solving sub problems (at least to some extent)
- Abstraction
  - abstraction is the separation of interface from internals
  - It is the finding of logical boundaries for modularity
  - Abstraction is the process of modularizing and then chunking components
  - About the property of modules being able to only consider the interface of a component rather than considering the internal implementations
  - Abstraction is a tool to reduce *granularity*
- Layering
  - One way of limiting or controlling how modules interact
  - Layered systems have modules which only communication with layers directly below or above them
  - Another way of creating lower granularity interfaces
  - Most programs use this to some degree to abstract APIs or tasks which have high numbers of repeated tasks
- Hierarchy
  - This is another way of organizing modules
  - Modules only interact in their subsystems
  - Subsystems only have a single module as their interface
  - The goal is to reduce the number of modules while allowing for more complex interconnections within subsystems
  - Limits the visibility of each level of abstraction to limit complexity (aids chunking)

Names Make Connections

- The four techniques decompose systems into smaller systems

- Naming modules allows us to begin recomposing small subsystems back into the larger system
- Naming allows for
  - postponing decisions (allowing for better decisions with more information)
  - Easy replacement of modules (pointing modules to different places)
  - Sharing of modules (code reuse, maybe even process reuse)

Unique Problem of Computer Systems

- The complexity is not limited by physical constraints
- Rate of technology change is unprecedented
- High hardware limits allow for extremely unprecedented complexity in systems, especially in systems which are made to take advantage of various types of scaling.

Coping with Complexity

- Deal with complexity via iteration
- Keep it simple
- By rapidly iterating we learn where the logical boundaries are, and how to best deliver constraints
- By keeping things simple we only add the minimum amount of abstraction, modularity, hierarchy, or layering, since each additional abstraction adds a constant amount of complexity.
- By getting feedback we can better define the requirements for systems in a sensible and practical manner, the goal is to build something reliable and useful, not to make something abstractly beautiful.

Rest of the Course

- How to use complexity management principles
  - Modularity
  - Abstractions
  - Client/Server and Virtualization

Fundamental Abstractions

1. Memory (objects by name)
   - Storage
   - Wide ranging technologies
     - Volitile vs non-Volitile
     - Latency vs cost
   - All memory has two fundamental operations
     - Write(name, value)
     - Read(name) $\rightarrow$ value
   - Memory devices read and write contiguous bits
   - Two Useful properties
     - Read/write coherence, read always returns the most recent write

    - Before or after atomicity, all read or writes behave as if they occured in a single linear order, ie: reads and writes have a cannonical order
  - Problems
    - Concurrency (non-linear or undefined access order)
    - Remote Storage (amplified delays in Reads/Writes)
    - Replicated Storage (to improve reliability at cost of synchronization)

2. Interpreter (manipulate names)
  - Active elements of computer, the things that do the computations
  - Wide-ranging and layered
    - Example: Human generates requests to calendar, Java interpreter, byte-code interpreter, hardware interpreter
  - Simple abstraction
    - Instruction reference, where is the next instruction
    - reprotoire, what actions are avalible
    - Environment, what is the current state?
  - Many systems have multiple interpreters
    - usually asynchronous
    - different progression rates
    - assume no control over exact order of execution
    - design alogithms to explicitly control timing and order

3. Communication (move data to and from names)
  - Allows for data movement between physically separate components
  - Fundamental interface
    - SEND(name, msg)
    - Recieve(name) $\rightarrow$ msg
  - Properties
    - Unpredictable time to complete SEND and RECIEVE
    - Hostile environment leads to data corruption
    - Data loss
    - Asynchronous timing, no garuntee of order