

System Design

An Engineering Approach to Computer Networking

What is system design?

- A computer network provides computation, storage and transmission resources
- System design is the art and science of putting together these resources into a harmonious whole
- Extract the most from what you have

Goal

- In any system, some resources are more freely available than others
 - ◆ high-end PC connected to Internet by a 28.8 modem
 - ◆ *constrained* resource is link bandwidth
 - ◆ PC CPU and memory are *unconstrained*
- Maximize a set of performance metrics given a set of resource constraints
- Explicitly identifying constraints and metrics helps in designing efficient systems
- Example
 - ◆ maximize reliability and MPG for a car that costs less than \$10,000 to manufacture

System design in real life

- Can't always quantify and control all aspects of a system
- Criteria such as scalability, modularity, extensibility, and elegance are important, but unquantifiable
- Rapid technological change can add or remove resource constraints (example?)
 - ◆ an ideal design is 'future proof'
- Market conditions may dictate changes to design halfway through the process
- International standards, which themselves change, also impose constraints
- Nevertheless, still possible to identify some principles

Some common resources

- Most resources are a combination of
 - ◆ time
 - ◆ space
 - ◆ computation
 - ◆ money
 - ◆ labor

Time

- Shows up in many constraints

- ◆ deadline for task completion
- ◆ time to market
- ◆ mean time between failures

- Metrics

- ◆ *response time*: mean time to complete a task
- ◆ *throughput*: number of tasks completed per unit time
- ◆ *degree of parallelism* = response time * throughput
 - ◆ 20 tasks complete in 10 seconds, and each task takes 3 seconds
 - ◆ => degree of parallelism = $3 * 20/10 = 6$

Space

- Shows up as
 - ◆ limit to available memory (kilobytes)
 - ◆ bandwidth (kilobits)
 - ◆ 1 kilobit/s = 1000 bits/sec, but 1 kilobyte/s = 1024 bits/sec!

Computation

- Amount of processing that can be done in unit time
- Can increase computing power by
 - ◆ using more processors
 - ◆ waiting for a while!

Money

- Constrains

- ◆ what components can be used
- ◆ what price users are willing to pay for a service
- ◆ the number of engineers available to complete a task

Labor

- Human effort required to design and build a system
- Constrains what can be done, and how fast

Social constraints

■ Standards

- ◆ force design to conform to requirements that may or may not make sense
- ◆ underspecified standard can faulty and non-interoperable implementations

■ Market requirements

- ◆ products may need to be backwards compatible
- ◆ may need to use a particular operating system
- ◆ example
 - ✦ GUI-centric design

Scaling

- A design constraint, rather than a resource constraint
- Can use any centralized elements in the design
 - ◆ forces the use of complicated distributed algorithms
- Hard to measure
 - ◆ but necessary for success

Common design techniques

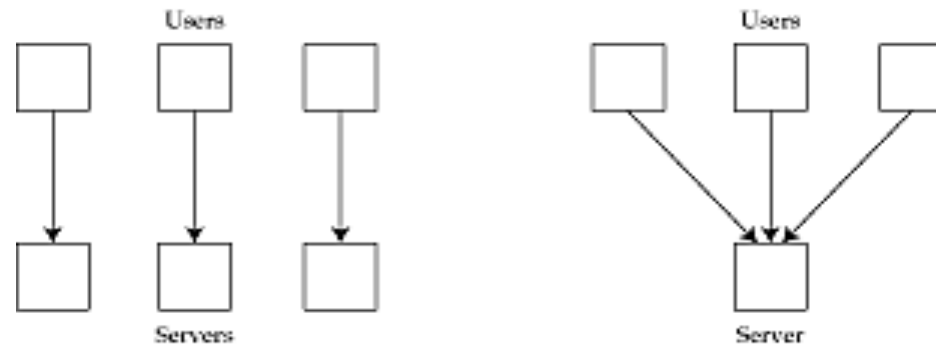
- Key concept: *bottleneck*
 - ◆ the most constrained element in a system
- System performance improves by removing bottleneck
 - ◆ but creates new bottlenecks
- In a *balanced* system, all resources are simultaneously bottlenecked
 - ◆ this is optimal
 - ◆ but nearly impossible to achieve
 - ◆ in practice, bottlenecks move from one part of the system to another
 - ◆ example: Ford Model T

Top level goal

- Use unconstrained resources to alleviate bottleneck
- How to do this?
- Several standard techniques allow us to trade off one resource for another

Multiplexing

- Another word for sharing
- Trades time and space for money
- Users see an increased response time, and take up space when waiting, but the system costs less
 - ◆ economies of scale



Multiplexing (contd.)

- Examples
 - ◆ multiplexed links
 - ◆ shared memory
- Another way to look at a shared resource
 - ◆ *unshared virtual resource*
- *Server* controls access to the shared resource
 - ◆ uses a *schedule* to resolve contention
 - ◆ choice of scheduling critical in proving quality of service guarantees

Statistical multiplexing

- Suppose resource has capacity C
- Shared by N identical tasks
- Each task requires capacity c
- If $Nc \leq C$, then the resource is underloaded
- If at most 10% of tasks active, then $C \geq Nc/10$ is enough
 - ◆ we have used statistical knowledge of users to reduce system cost
 - ◆ this is *statistical multiplexing gain*

Statistical multiplexing (contd.)

- Two types: spatial and temporal
- Spatial
 - ◆ we expect only a fraction of tasks to be simultaneously active
- Temporal
 - ◆ we expect a task to be active only part of the time
 - ✦ e.g silence periods during a voice call

Example of statistical multiplexing gain

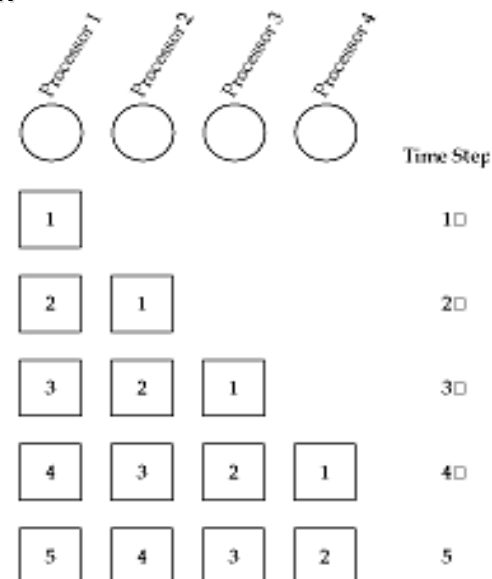
- Consider a 100 room hotel
- How many external phone lines does it need?
 - ◆ each line costs money to install and rent
 - ◆ tradeoff
- What if a voice call is active only 40% of the time?
 - ◆ can get both spatial and temporal statistical multiplexing gain
 - ◆ but only in a packet-switched network (why?)
- Remember
 - ◆ to get SMG, we need good statistics!
 - ◆ if statistics are incorrect or change over time, we're in trouble
 - ◆ example: road system

Pipelining

- Suppose you wanted to complete a task in less time
- Could you use more processors to do so?
- Yes, if you can break up the task into *independent* subtasks
 - ◆ such as downloading images into a browser
 - ◆ optimal if all subtasks take the same time
- What if subtasks are dependent?
 - ◆ for instance, a subtask may not begin execution before another ends
 - ◆ such as in cooking
- Then, having more processors doesn't always help (example?)

Pipelining (contd.)

- Special case of *serially dependent* subtasks
 - ◆ a subtask depends only on previous one in execution chain
- Can use a *pipeline*
 - ◆ think of an assembly line



Pipelining (contd.)

- What is the best decomposition?
- If sum of times taken by all stages, Response Time = R
- Slowest stage takes time S
- Throughput = $1/S$
- Degree of parallelism = R/S
- Maximize parallelism when $R/S = N$, so that $S = R/N \Rightarrow$ equal stages
 - ◆ *balanced pipeline*

Batching

- Group tasks together to amortize overhead
- Only works when overhead for N tasks $<$ N time overhead for one task (i.e. *nonlinear*)
- Also, time taken to accumulate a batch shouldn't be too long
- We're trading off reduced overhead for a longer worst case response time and increased throughput

Exploiting locality

- If the system accessed some data at a given time, it is likely that it will access the same or 'nearby' data 'soon'
- Nearby => spatial
- Soon => temporal
- Both may coexist
- Exploit it if you can
 - ◆ caching
 - ✦ get the speed of RAM and the capacity of disk

Optimizing the common case

- 80/20 rule
 - ◆ 80% of the time is spent in 20% of the code
- Optimize the 20% that counts
 - ◆ need to measure first!
 - ◆ RISC
- How much does it help?
 - ◆ Amdahl's law
 - ◆ Execution time after improvement = (execution affected by improvement / amount of improvement) + execution unaffected
 - ◆ beyond a point, speeding up the common case doesn't help

Hierarchy

- Recursive decomposition of a system into smaller pieces that depend only on parent for proper execution
- No single point of control
- Highly scaleable
- Leaf-to-leaf communication can be expensive
 - ◆ shortcuts help

Binding and indirection

- Abstraction is good
 - ◆ allows generality of description
 - ◆ e.g. mail aliases
- Binding: translation from an abstraction to an instance
- If translation table is stored in a well known place, we can bind automatically
 - ◆ indirection
- Examples
 - ◆ mail alias file
 - ◆ page table
 - ◆ telephone numbers in a cellular system

Virtualization

- A combination of indirection and multiplexing
- Refer to a virtual resource that gets matched to an instance at run time
- Build system as if real resource were available
 - ◆ virtual memory
 - ◆ virtual modem
 - ◆ Santa Claus
- Can cleanly and dynamically reconfigure system

Randomization

- Allows us to break a tie fairly
- A powerful tool
- Examples
 - ◆ resolving contention in a broadcast medium
 - ◆ choosing multicast timeouts

Soft state

- State: memory in the system that influences future behavior
 - ◆ for instance, VCI translation table
- State is created in many different ways
 - ◆ signaling
 - ◆ network management
 - ◆ routing
- How to delete it?
- Soft state => delete on a timer
- If you want to keep it, refresh
- Automatically cleans up after a failure
 - ◆ but increases bandwidth requirement

Exchanging state explicitly

- Network elements often need to exchange state
- Can do this implicitly or explicitly
- Where possible, use explicit state exchange

Hysteresis

- Suppose system changes state depending on whether a variable is above or below a threshold
- Problem if variable fluctuates near threshold
 - ◆ rapid fluctuations in system state
- Use state-dependent threshold, or *hysteresis*

Separating data and control

- Divide actions that happen once per data transfer from actions that happen once per packet
 - ◆ Data path and control path
- Can increase throughput by minimizing actions in data path
- Example
 - ◆ connection-oriented networks
- On the other hand, keeping control information in data element has its advantages
 - ◆ per-packet QoS

Extensibility

- Always a good idea to leave hooks that allow for future growth
- Examples
 - ◆ Version field in header
 - ◆ Modem negotiation

Performance analysis and tuning

- Use the techniques discussed to tune existing systems

- Steps

- ◆ measure
- ◆ characterize workload
- ◆ build a system model
- ◆ analyze
- ◆ implement

