Lecture 2 – Introduction to System Design

Outline

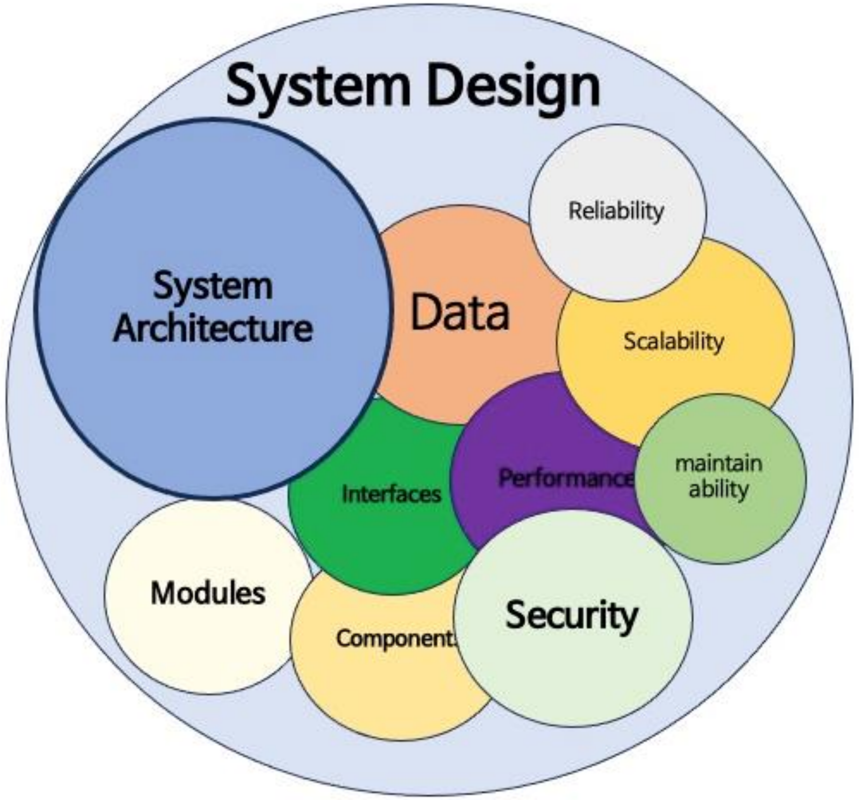
What is system design?	System design patterns:
• Optimisation • Bottleneck • Constraint • Resource • metric	• Multiplexing • Pipelining • Batching • Locality/Caching • Distributed/Hierarchy • Abstraction/Binding and Indirection • Virtualisation / Randomisation • Data and control separation

What is system design?

- A **system** is a collection of **components** (such as hardware and software) that work together to achieve a **specific goal**.
  - A computer consists of CPU, memory, devices
  - A network consists of end-hosts, links, switches, routers

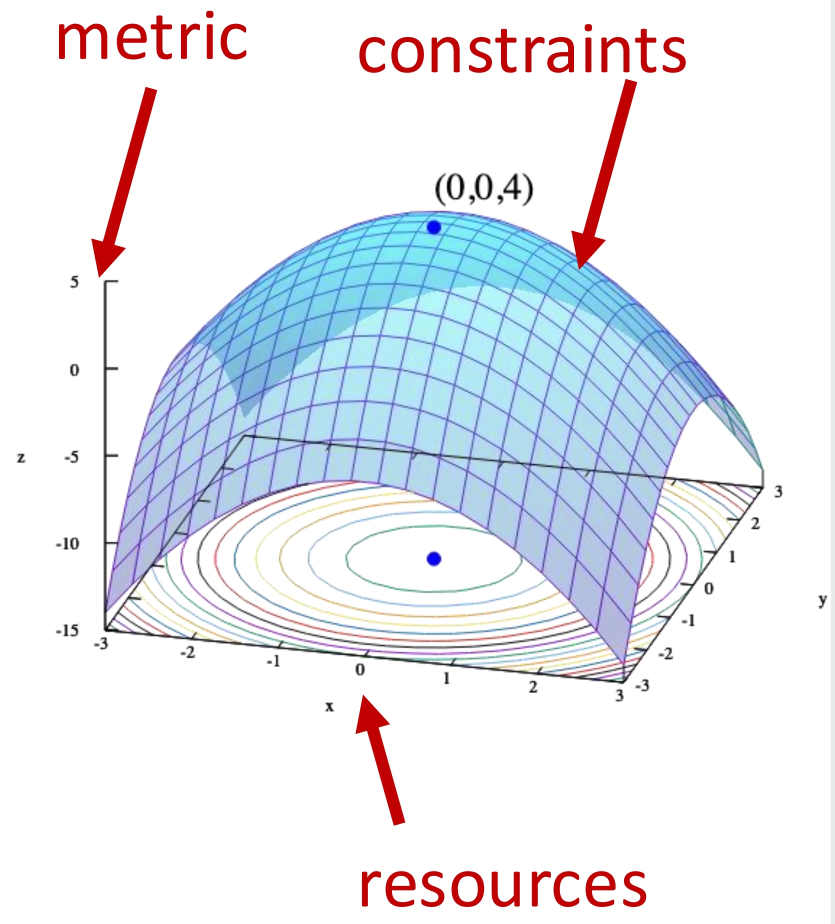
- ***System design*** is the science (and a bit of art) of putting together these resources into a harmonious whole
  - *Extract the most from what you have (optimisation)*

- *In our module, we will explore how system design techniques can be applied in OS, networking and the cloud.*



System Optimization

- In any system, some resources are more freely available than others
    - High-end PC connected to the Internet via a poor and noisy WiFi link
    - *The constrained* resource is link bandwidth
    - PC, CPU, and memory are *unconstrained*
- Optimization: Maximize a set of performance metrics (e.g., time to load webpage) given a set of resource constraints (e.g. bandwidth)
- Explicitly identifying *metrics*, *resources,* and *constraints* helps in designing efficient systems



Common Resources (1)

- **Time**: can be translated into multiple aspects
  - deadline for task completion
  - time to market
  - mean time between failures
- **Space**: Shows up as
  - Number of PCI slots
  - limit to available memory (kilobytes)
  - bandwidth (kilobits)
    - 1 kilobit/s = 1000 bits/sec, but 1 kilobyte/s = 1024 bits/sec!

Common Resources (2)

- **Computation**: Amount of processing that can be done in unit time
  - Can increase computing power by
    - using more processors
    - waiting for a while!
- **Cost**: cost to implement system
  - 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

Constraints: Scaling

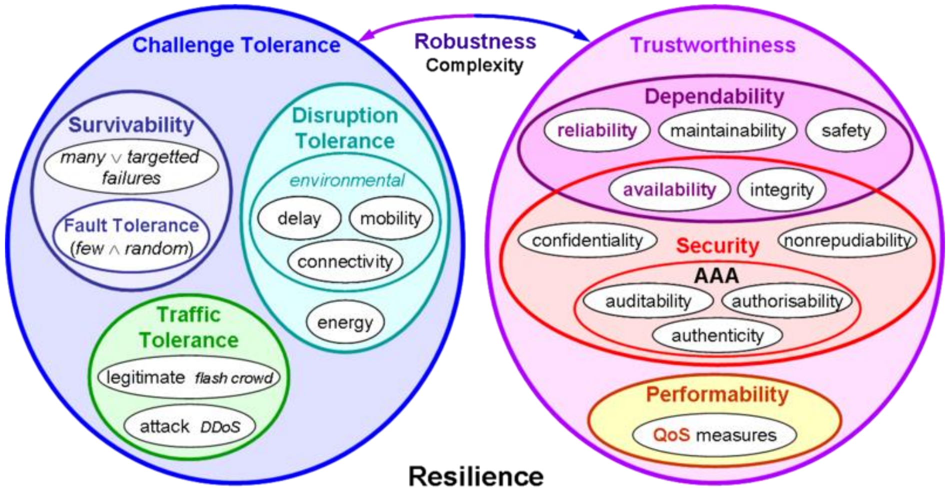
- **Scaling** is the ability of a system to handle an increasing amount of work, users, or data by adding resources (such as CPU, memory, bandwidth, or servers) without a proportional loss of performance.
  - A design constraint, rather than a resource constraint
- Can use any centralized elements in the design
  - forces the use of complicated distributed algorithms
- Hard to measure
  - but necessary for success

Constraints: Social

- **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

Constraints: Resilience

- **Resilience** is the ability of a system to **continue operating correctly, or recover quickly**, when it faces unexpected problems such as hardware failures, software bugs, high load, or cyberattacks
  - Includes quantitative (availability) and qualitative aspects (safety)
- Typically, resilience depends on risk analysis/threat model to understand constrains.

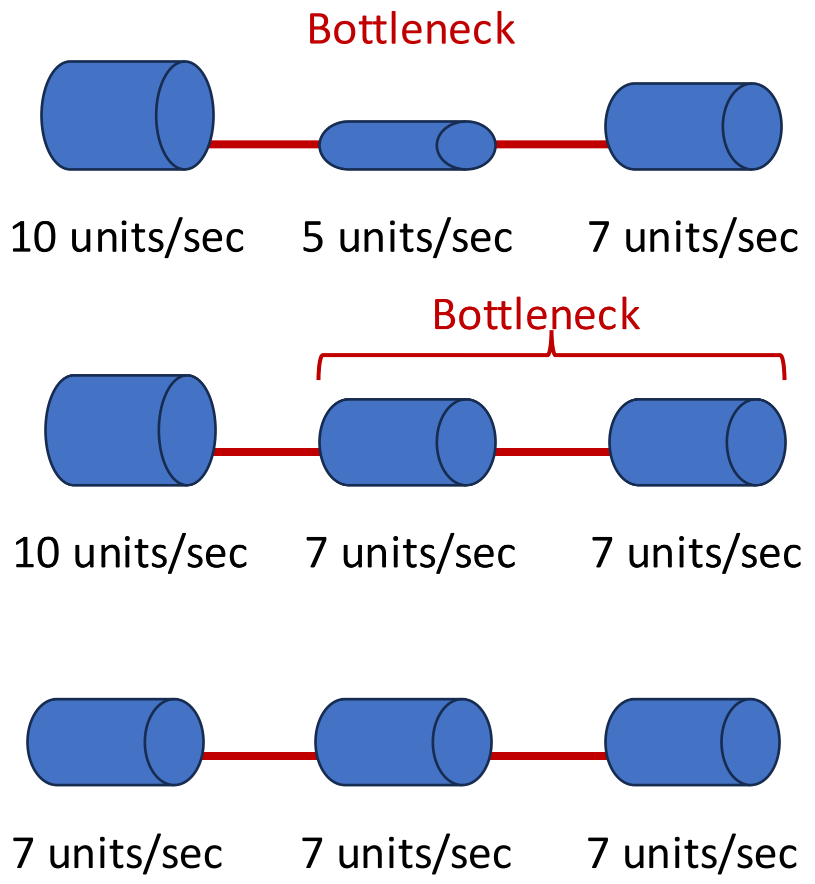


Metrics

Category	Goal / What It Measures	Typical Metrics
Performance / Efficiency	Speed or responsiveness of the system	**Latency** (response time) **Throughput** (ops/sec) **Utilization** (CPU, link)
Scalability	How performance changes with workload or resources	**Elasticity** (load adaptation) **Parallel efficiency**
Reliability / Availability / Resilience / Security	Ability to function despite failures	**MTBF (Mean Time Between Failures)** **Availability = Uptime/(Uptime+Downtime)** **Isolation strength**
Cost / Resource Utilization	Efficiency in using limited resources	**CapEx / OpEx- Memory /Network bandwidth usage**
Maintainability / Evolvability	Ease of modification, debugging, or scaling	**Code complexity (LOC, cyclomatic)** **Deployment time - Configuration overhead**
Sustainability	Environmental, power and material impact	**Energy per operation** **Carbon footprint (CO2 mass)**

System Bottlenecks

- A ***bottleneck**, is* the most constrained element in a system with respect to our optimization metric (e.g. longest/slowest process)
- Performance metrics improves by removing system bottlenecks
  - This process can create 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

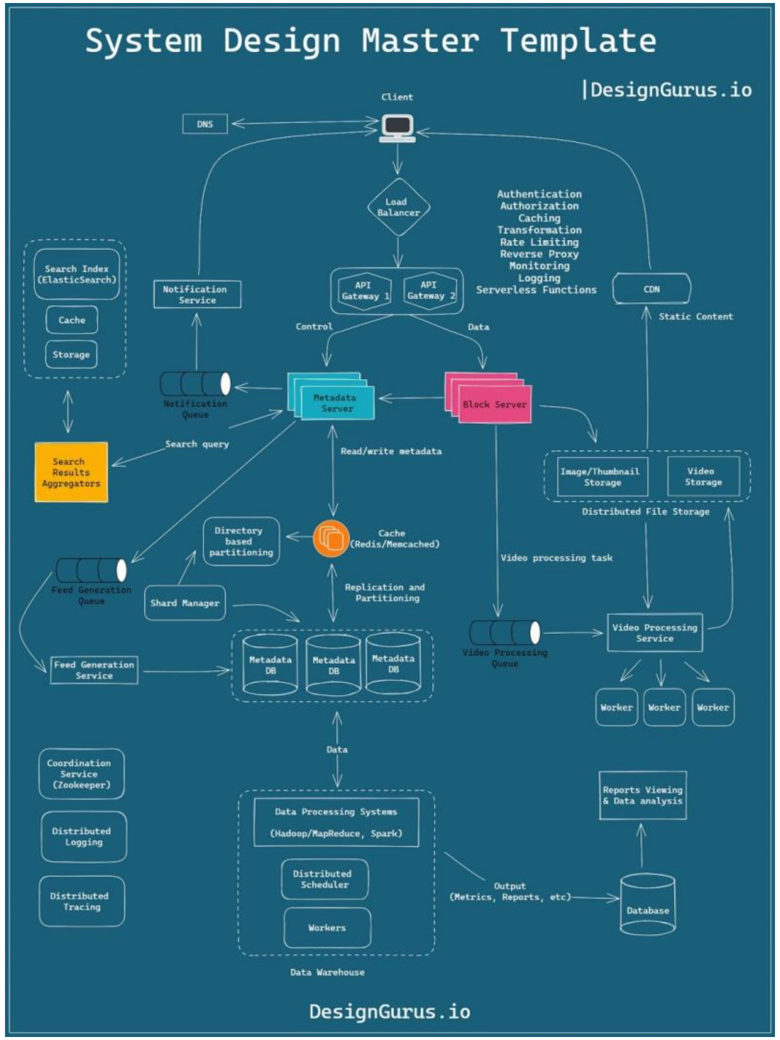


System Design in Real Life

- Can't always quantify and control all system metrics
  - Metrics such as *scalability*, *modularity*, *extensibility*, and *elegance* are essential, but unquantifiable
- Rapid technological change can add or remove resource constraints
  - Multi-core CPUs can overcome the limitations of CPU clock speed, but need a rethink for application architectures
- Market conditions may dictate changes to the design halfway through the process
- Nevertheless, still possible to identify some principles

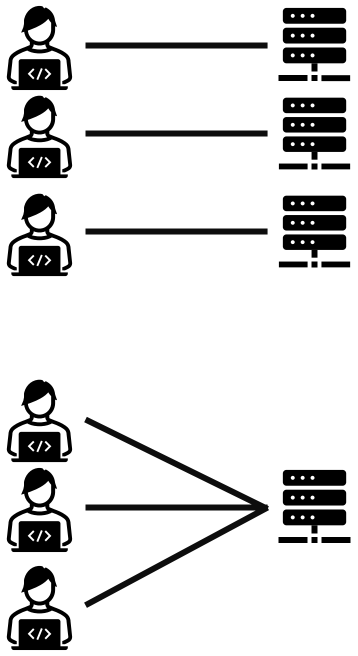
System Desing Patterns

- Use unconstrained resources to alleviate bottleneck
- Several **standard design patterns** allow us to trade off one resource for another
- *We will revisit these theoretical concepts and explain how they apply in different scenarios in OS and network design.*



Multiplexing

- Use the same resource to service multiple requests without interference
  - Another word for sharing
- Goal: serve a large number of users, without increasing resource requirements.
  - Excellent for coping with **cost** constraints
  - Users see an increased response time, and take up space when waiting, but the system costs less
  - economies of scale



Multiplexing examples

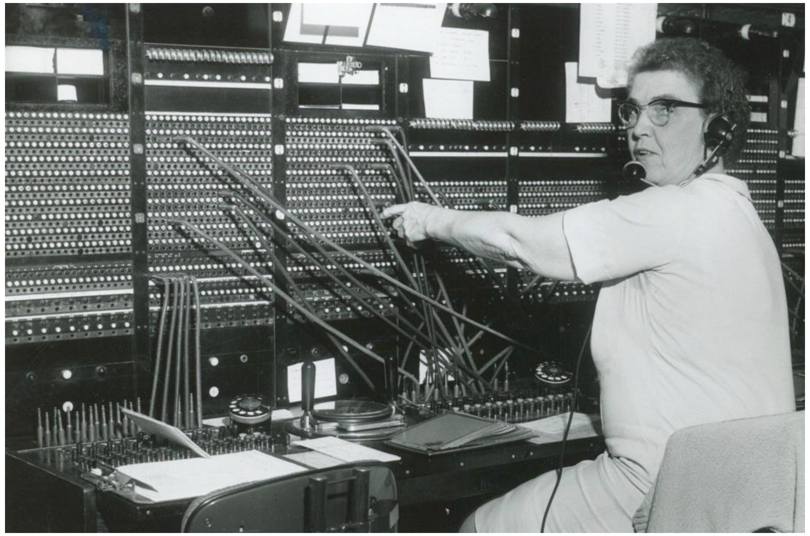
- Multiplexing dimensions:
  - Time: CPU and I/O multiplexing
    - Multiple processes share the CPU and read/write on the same disk
    - OS scheduler ensure processes do not interfere
  - Space: Memory multiplexing
    - Virtual memory gives each process its own address space
  - Frequency: Radio comms
    - Mobile networks use the same airwaves to transmit signal.
    - Complex algorithms use multiple frequencies to increase user number, without affecting performance and resilience.

Statistical Multiplexing

- Share resources dynamically based on demand.
- Suppose the resource has capacity C
- Shared by N identical tasks
- Each task requires capacity c
- If Nc <= C, then the resource is underloaded
- If at most 10% of tasks are active, then C >= Nc/10 is enough
  - We use the statistical knowledge of users to reduce system cost
  - This is *the statistical multiplexing gain*

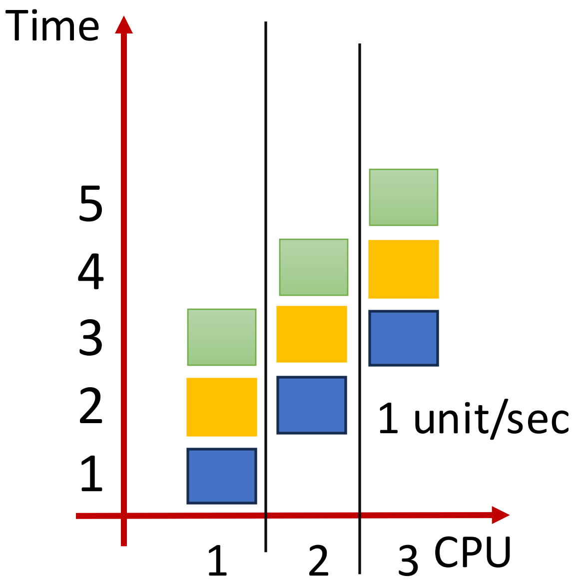
Example of statistical multiplexing gain

- Consider a 100-room hotel
- How many external phone lines does it need?
  - Trade-off: each line costs money to install and rent
- What if a voice call is active only 40% of the time?
  - Rent only 40 lines, drop call if > 40 simultaneous calls
  - Negatively impact on user satisfaction, if peak load estimation is low
- Good statistics on resource use are essential to achieve gains
  - If statistics are incorrect or change over time, we're in trouble
  - Example: road system



Pipelining (1)

- Suppose you have a time constraint; i.e. 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
  - Example: downloading page objects in 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.



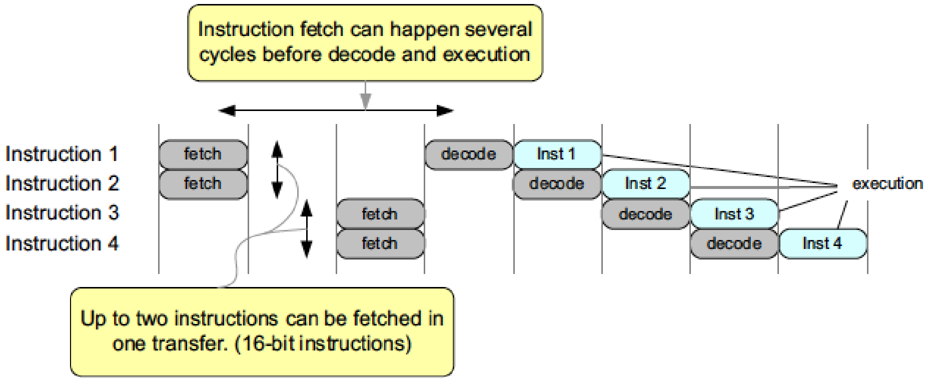
Pipelining (2)

- Special case of *serially dependent* subtasks
  - A subtask depends only on the previous one in the execution chain
- Can use a *pipeline*
  - think of an assembly line
- In pipelining, multiple **stages** overlap during execution.
  - Each processor works on a different stage of the process.

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

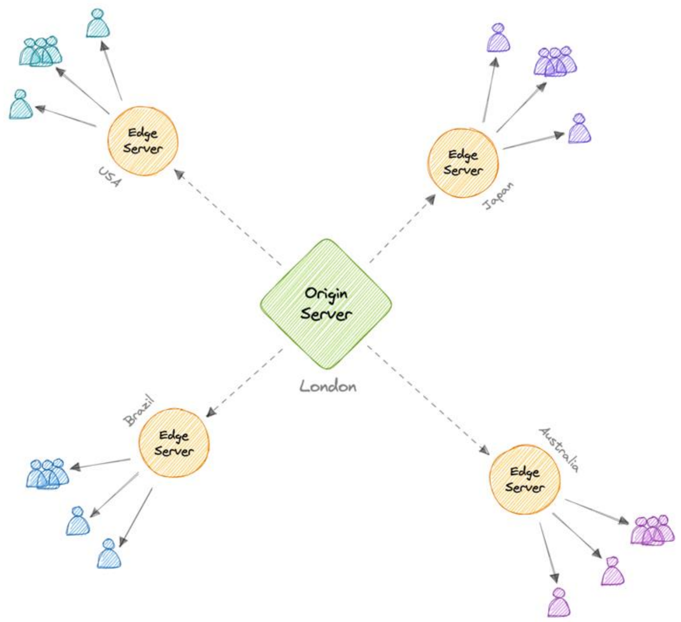
Batching

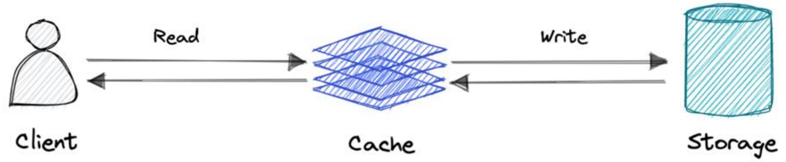
- Group tasks together and process them as a single unit, rather than handling them individually.
  - Amortise overheads (e.g.. loading time)
- Only works when `*overhead for N tasks*` < `*N time overhead for one task*` (i.e. *nonlinear*)
  - If loading two 16-bit instructions takes the same time to load 1 16-bit instruction, then you improved read access times.
- Also, the 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
  - Improve average response **time** of the system.
  - Improve CPU constraints; Complete more instructions per second.



Exploiting Locality

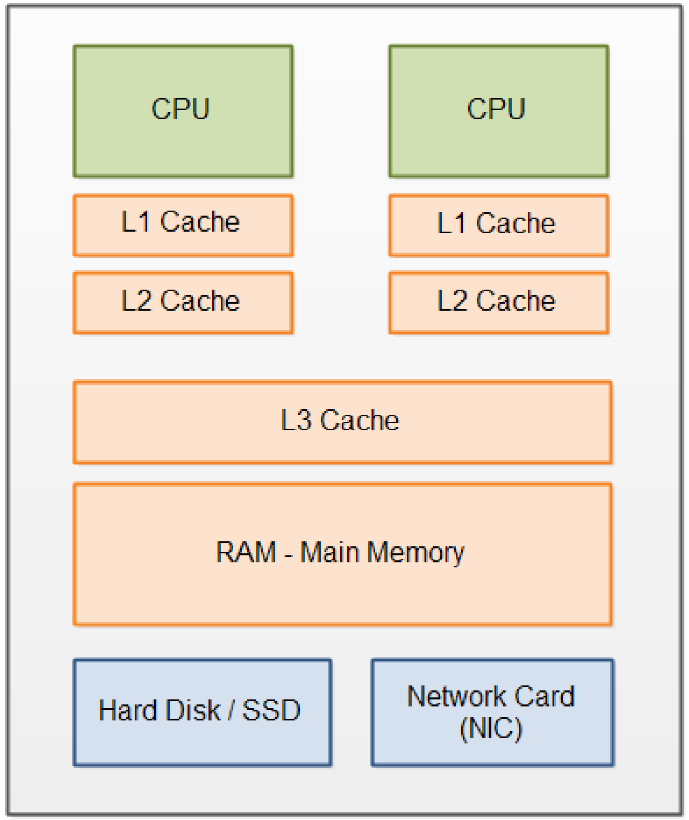
- If the system accessed some data at a given time, it is likely that it will access the same or ‘nearby’ data ‘soon’
  - Nearby => spatial, e.g., watching a new episode on iPlayer in the UK
  - Soon => temporal, e.g., next instruction in a program
- Improve access time constraints, reduce load on paths to main server.
- Caching: store copies of frequently accessed data from a **larger, slower storage** in **small, fast storage instances**.
  - Example: Filesystem caching - get the speed of RAM and the capacity of disk





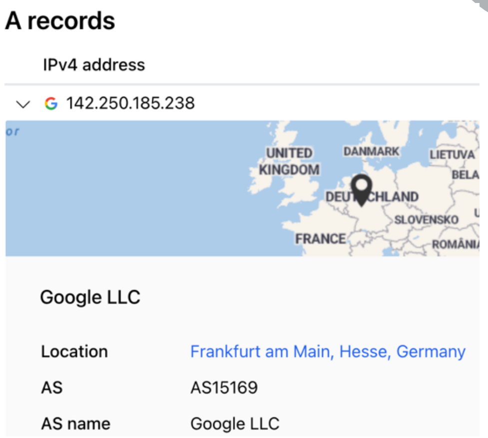
Hierarchy

- Recursive decomposition of a system into smaller pieces that depend only on the parent for proper execution
- Examples: Computer memory, Addressing in Networks
- No single point of control
- Highly scalable
- Leaf-to-leaf communication can be expensive
  - shortcuts help

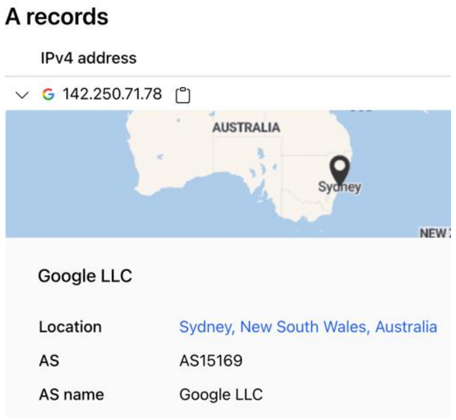


Binding and indirection

- **Abstraction** is good
  - Allows generality of description: Domain names, variable pointers
- **Binding**: translate an abstraction to an instance
- **Indirection**: use a reference to **access something**, instead of directly
  - Provides **flexibility** because the reference can change without affecting the higher-level system
  - Dynamic binding allows easy optimisation for current conditions
    - E.g., use domain-name to get the closest instance of a service
- Common technique in the Internet; can direct user to the best server based on current conditions



https://www.nslookup.io/domains/google.com/dns-records/#thenetherlands



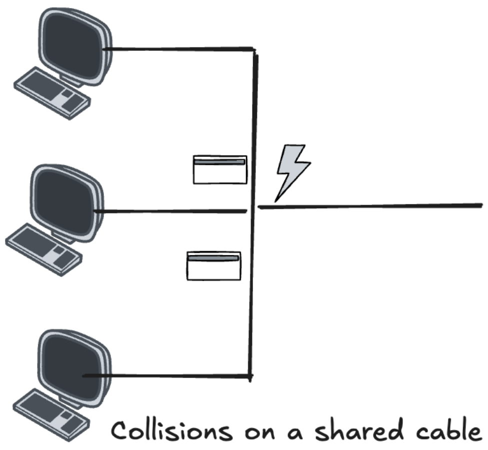
https://www.nslookup.io/domains/google.com/dns-records/#australia

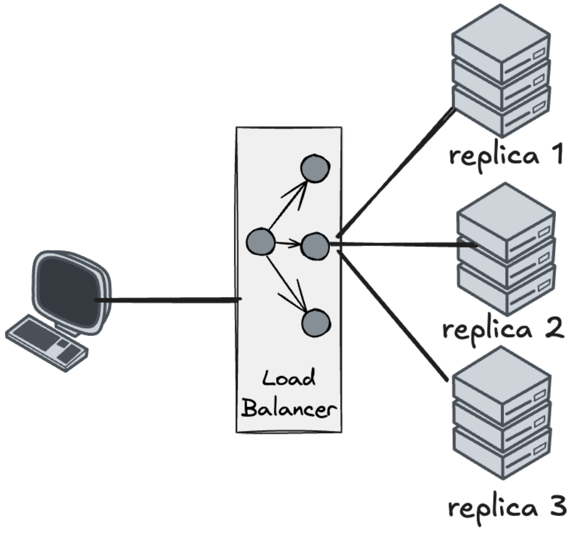
Virtualization

- A combination of indirection and multiplexing
- Create a **virtual version of a resource** (hardware, OS, network, or storage) so that multiple independent systems or applications can share it as if each has its own dedicated resource
- Examples:
  - virtual memory
  - virtual modem
  - Cloud computing
  - Santa claus
- Can cleanly and dynamically reconfigure the system
- Simplifies programming for developers; you can code without worrying synchronization with other programs

Randomization

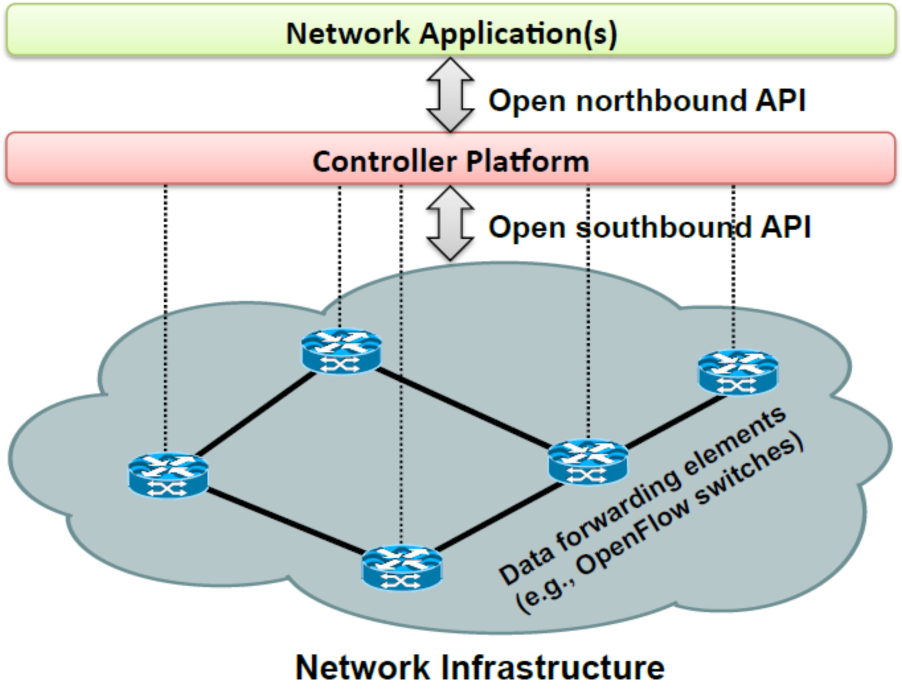
- Allows us to break a tie fairly
- Can also be used as a simple mechanism to spread load
- Examples
  - resolving contention in a broadcast medium
  - Use hashes on load balancers to spread traffic across backend servers.
- Statistics are important.
  - If task effort is not the same, fairness can be compromised





Separating data and control

- Divide actions that happen once per data transfer (control plane) from actions that occur once per packet (data plane)
    - **Data plane** = *"what is being carried"* (payload, instructions, user content).
    - **Control plane** = *"how and where it should go"* (rules, scheduling, routing).
- Can increase throughput by optimizing processing in the data path
- Can improve intelligence through the data plane
- Example
    - Software-defined networks
- On the other hand, keeping control information in data plane can improve optimisation
    - per-packet QoS



Performance analysis and tuning

- Use the techniques discussed to tune existing systems
- Steps
  - measure
  - characterize workload
  - build a system model
  - analyse
  - implement

Conclusions

- System design is the science of combining multiple components to deliver specific functionalities.
  - Optimize a metrics, by using resources and meeting constraints.
- We use design patterns to improve the performance of a system.
  - Explore system design patterns in action as we discuss in more depth networking and OS technologies.
- Next Topic: Operating Systems and Processes

Labs start next week, check your calendar!!!!

Questions (1)

- From your experience with everyday life, can you find examples that use system design techniques to improve system performance?
  - E.g. cashiers in the supermarket multiplex queues, TAs in the lab allow support indirection

Question (2)

- Assume that Database1 and Database2 serve the same static content to different parts of the network. Can you think of an application of a design pattern in the architecture below to improve the system cost? What impact does your design have on the system security?