Web Application Performance and Scalability

Definitions

Scalability Measures, Architecture, and Optimizations

Performance Measures, Architecture, and Optimizations

DEFINITIONS

Definitions

Quality of Service (QoS) – Defined w.r.t. a performance measure to indicate the acceptable level of that service

Service-Level Agreement (SLA) – A (contractual) guarantee to meet various performance criteria under defined operating constraints

Resources – A service tied to physical env; RAM, disk, network, CPU, etc.

- Important because resources are limited, so there may be *contention*

Profiling – A process to understand the Resources consumed by, or QoS of, one or more application components

Bottleneck – A component that *performs* the poorest w.r.t. other components (slowest, consumes the most resources, etc.)

Benchmarking – Established baselines for comparative analysis

Baseline – Like a benchmark, establishes a known point in the measurement of the system; changes are then compared against it

Clustering – The ability to allocate more resources (usually entire servers) and maintain the interface of a single logical application

Elasticity – The ability to expand/contract resource provision on demand

SCALABILITY

Scalability

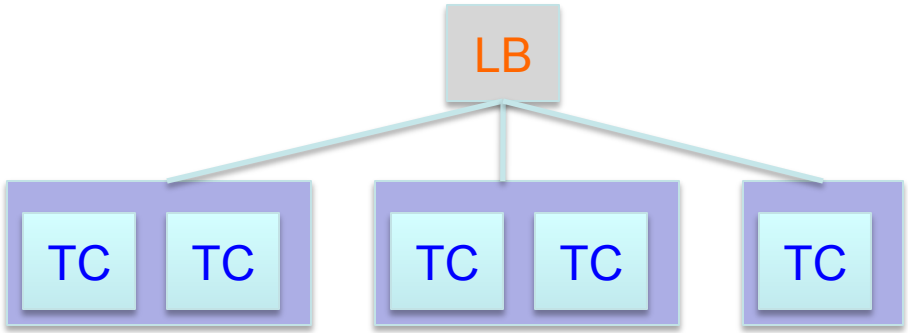
Scalability is the ability to maintain near-constant *Quality of Service* (QoS) in the face of different loads

- One way people think about it: The app should *gracefully degrade* in the face of increasing demand under constant resources
  - This is a *fault-tolerant viewpoint*; the system should handle burst traffic or stress periods without failure
- Another way: The ability to add resources to a deployed environment to handle increasing load.
  - This is an *architectural viewpoint*; the system should be designed in such a way that an organization can allocate resources according to expected need.
- Taken together, these two perspectives (broadly interpreted) motivate *virtualization services, cloud computing, & containerization*
  - In this approach, resources are allocated on demand based on traffic patterns. Capacity models are created and resources allocated under a *service model* for which providers assign a fee

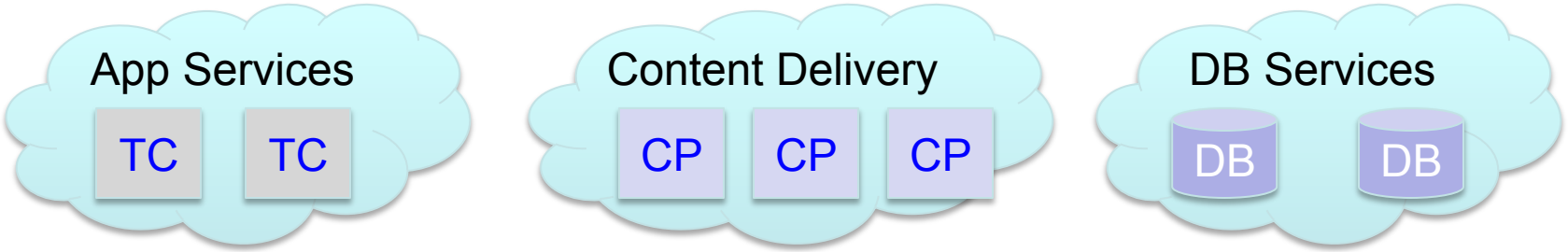
Scalability Optimizations

Scalability Patterns: 5 patterns to help with QoS attributes

- **Pattern 1: Replication**
  - Replicate the web application across several container instances
  - Container instances mapped to many computational “nodes”
  - Most common
  - Issues:
    - Statefulness
    - Shared Resources



- **Pattern 2: Clustering**
  - Ability to assign multiple computing resources to a single logical service



- **Pattern 3: Horizontal and Vertical Fragmentation**

Pattern 3: Horizontal/Vertical Fragmentation

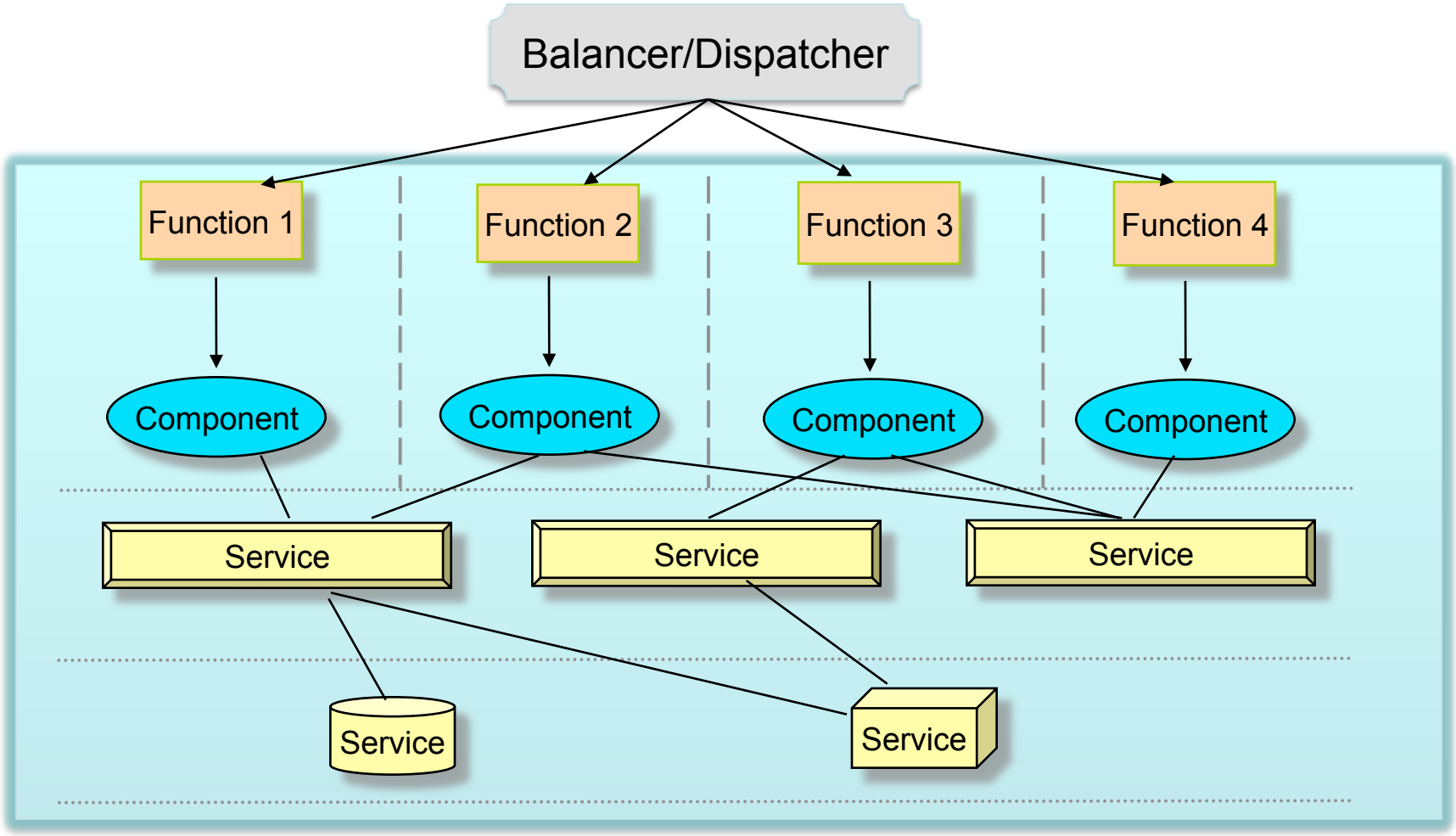
A common mistake is to envision many *applications* as <u>one</u>

- Many web *systems* are collections of applications that are logically cohesive but architecturally decouple-able
- The decoupling can be identified from one or more *perspectives*
  - User perspective – different types of user scenarios and permissions
  - Functional perspective – different types of operations
- Example: eLearning Application
  - User types: Students, Professors, TAs, Advisors
  - Functional types: “Register”, “Submit Assignment”, “Post Grades”, “Waive Prerequisite”, “Approve POS”

Vertical Fragmentation is the design process of

- Identifying these perspectives
- Componentizing the system to allow for dedicated *optimization strategies* to be applied to each vertical area
  - Can often be a benefit in software evolution and maintenance as well
- Cohesiveness is achieved by sharing usability metaphors and visual elements (branding, skinning)

Combining Horizontal & Vertical Fragmentation



Vertical: Assign function responsibilities to components

Horizontal: Components utilize service layer(s)

Pattern 4: Virtual Machines & Containers

We've defined *elasticity* – why is it important?

- We talk a lot about perf/scale measures and technology drivers
- *What about cost?* Tradiationally operations creates a cost model based on cost per transaction (throughput), per request, per CPU, etc.
- Wasted capital costs are measured by computing resource idle time
- What if we could buy the computing resources as we need them?

Cloud Computing

- Supports elasticity by providing computing resources on demand
- You hear of 3 common models: SaaS, PaaS, and IaaS ← we are talking IaaS
- We achieve cost benefits in the following ways:
  - Reduced TCO for purchasing and physically housing iron
  - Pay-as-you-need computing resources, and an ability to spin up new VMs/containers
  - The containerization benefits of speed we discussed – avoiding the Matrix of Hell, having dev teams support products, continuous*

The ability of your Chief Operations Officer to select the right IaaS (or go old-school in iron or rented racks) is a key Factor to your organization's success!

Pattern 5: Maximizing Client Processing

MVC is shifting from the server more and more to the client

- More web applications rely on client-side processing
  - Javascript, AJAX, JSON, JQuery, CSS, HTML5, Flash, etc.
- The server provides ReSTful services
  - The more stateless your server is the better it will perform and scale
- This is clearly good for the server
  - It can serve a lot of stateless requests
- Is it so good for the client?
  - Initially these technologies enhanced the user experience
  - But other issues can come up:
    - Differences in client-side platform processing – need your standards!
    - Heavier processing burden on the client means it is harder to manage performance expectations
    - New class of security concerns (XSS, spoofing, etc.)

What a resource bargain for server-side provisioning – use the client's Computational resources (RAM, CPU) = lots more traffic I can handle!

PERFORMANCE

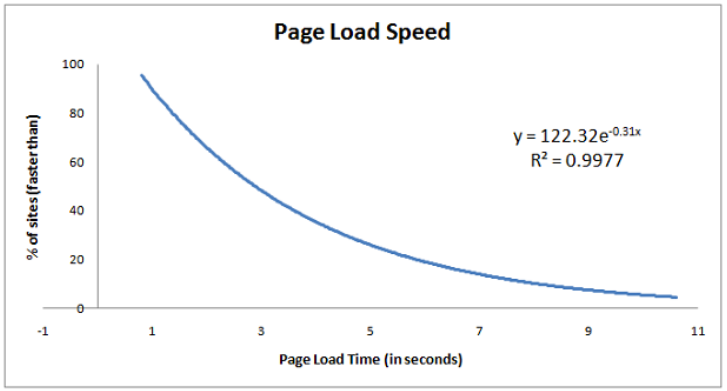
Performance Measures

Performance is evaluated w.r.t. observable characteristics

- Response time, user satisfaction, resource utilization, throughput
- Note that *qualitative* measures may be applicable too

Response Time Measure – the "Classic Measure"

- Measured in seconds/milliseconds via a client/server tool
- Page "fully loaded" times vary by mobile vs. desktop
  - Graph to the right shows ¾ page loads < 5s, ½ < 3s, ¼ < 1.7s
  - A February 2017 Google report cpncludes that the probability of a mobile user leaving the site increases to 32% @ 3s, 90% @ 5s, and 123% @ 10s
- Conclusion: Load a page in < 4s!



Kenyon, 2011

Data Collection and Analysis

- Performance sensitive to operational env
  - Resources (CPU, memory, bandwidth speed) on both client & server affect overall UX
- "Ideal" benchmark – assuring no resource contention
- "Typical use" benchmark – Under expected load/conditions

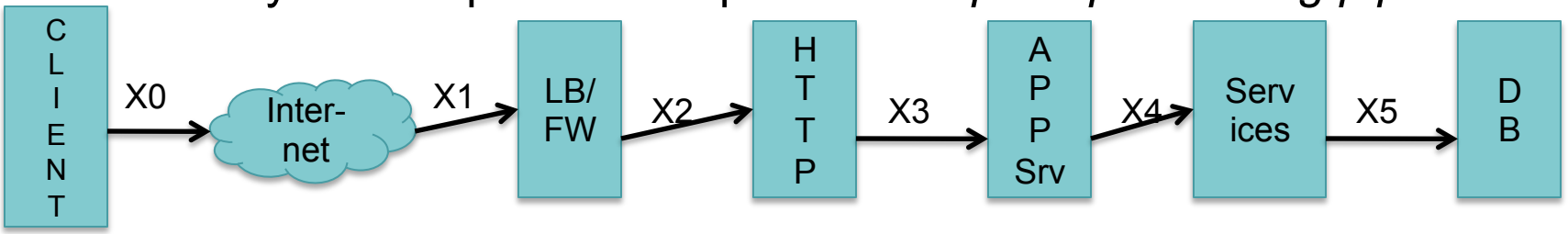
Performance Drill-Down

Foundational Concept: Queuing Theory

- From Operations research – the study of "waiting in line"
  - Applications in job-shop scheduling, network routing of packets
- For web, model request processing through a layered architecture
  - Little's Law: # of requests in process = avg intake * avg processing time

Example 2: Resources pooling

- Think of your components as part of a *request processing pipeline*:



- What is the % of requests through the pipeline?
  - In other words, the ratio between X_i and X_{i+1}?
  - At any stage, you can have requests "stacked up" due to an imbalance between requests and capacity ← BOTTLENECK!

Tuning: *a process optimizing each component architecture against expected traffic – This is a post-arhictecture, data-driven, configuration exercise!*

Performance vs. Scalability

Performance and Scalability Overlap

- Better performance also produces better scalability if:
  - The performance optimizations reduce resource consumption
  - Optimizations do not introduce errors at higher loads
- Better performance produces worse scalability if:
  - Design optimizations introduce coupling
  - Optimizations require increased resource consumption
    - » One example: caching and pooling often “hog” resources

Performance and Scalability Tips

- Design for it! Component-based modular design allows you to optimize (perf) & dynamically allocate resources (scale)
- Tune your application! Tuning can result in tremendous gains in performance and scalability for relatively little effort.
  - Many application support people do not have the skillset nowadays!
- Test for it! Throughout the lifecycle
  - Include early-stage profiling and analysis activities
  - Create/conduct benchmarks at the component and system levels

Testing for Performance and Scalability

Types of tests:

1. *Profiling test* – Measuring resource consumption or response time on a per page basis under ideal (unlimited) resource conditions
   - Goal: determine the application's baseline "footprint"
2. *Typical use test* – Measuring performance under typical loads with a full cross-section of user types
   - Goal: Most commonly used for response time validation
   - Difficult to use in troubleshooting as problems are difficult to pinpoint
3. *Peak use test* – Like typical use, except mapped to highest expected usage scenarios
   - Same goals and issues as above
4. *Stress test* – Exercise the application beyond its maximum anticipated load
   - Goal: Check for graceful degradation and breaking points
5. *Duration test* – Exercise the application at normal or below normal usage loads for an extended period
   - Goal: Check for resource leaks and other cumulative problems

Testing for Performance and Scalability

A common *faux pas* – optimizing without knowing your application's behavior!

- Optimization, at any stage in the develop/deploy process, should be *data-driven!*
- Follow the "20/80" rule for optimization – optimize the 20% of the application that is causing 80% of your performance problems!

A Top 5 List for Performance & Scalability Testing

1. Do it early as part of analysis – each developer should know how to perf test her/his application components
2. Do it early as part of iterative development – dev teams should do it as part of iteration closeout activities
3. Know your testing goal – idealized use? production benchmark?
4. Invest time in perf/scale testing infrastructure – common tools, common scripts, code instrumentation, test data repositories
5. Create benchmarks and use in regression tests