

DEPAUL UNIVERSITY

SE480 Week 4 – Testability, Interoperability, Performance

Steven Engelhardt

DePaul University

Autumn 2020

Table of Contents

Table of Contents

Summary of Last Week

- *Security* is about preventing unauthorized access while still allowing authorized access
  - Architecting for security often requires a fundamental knowledge of *cryptography*
  - More than any other quality, security benefits from *consulting with specialists* early and frequently

The Importance of Security

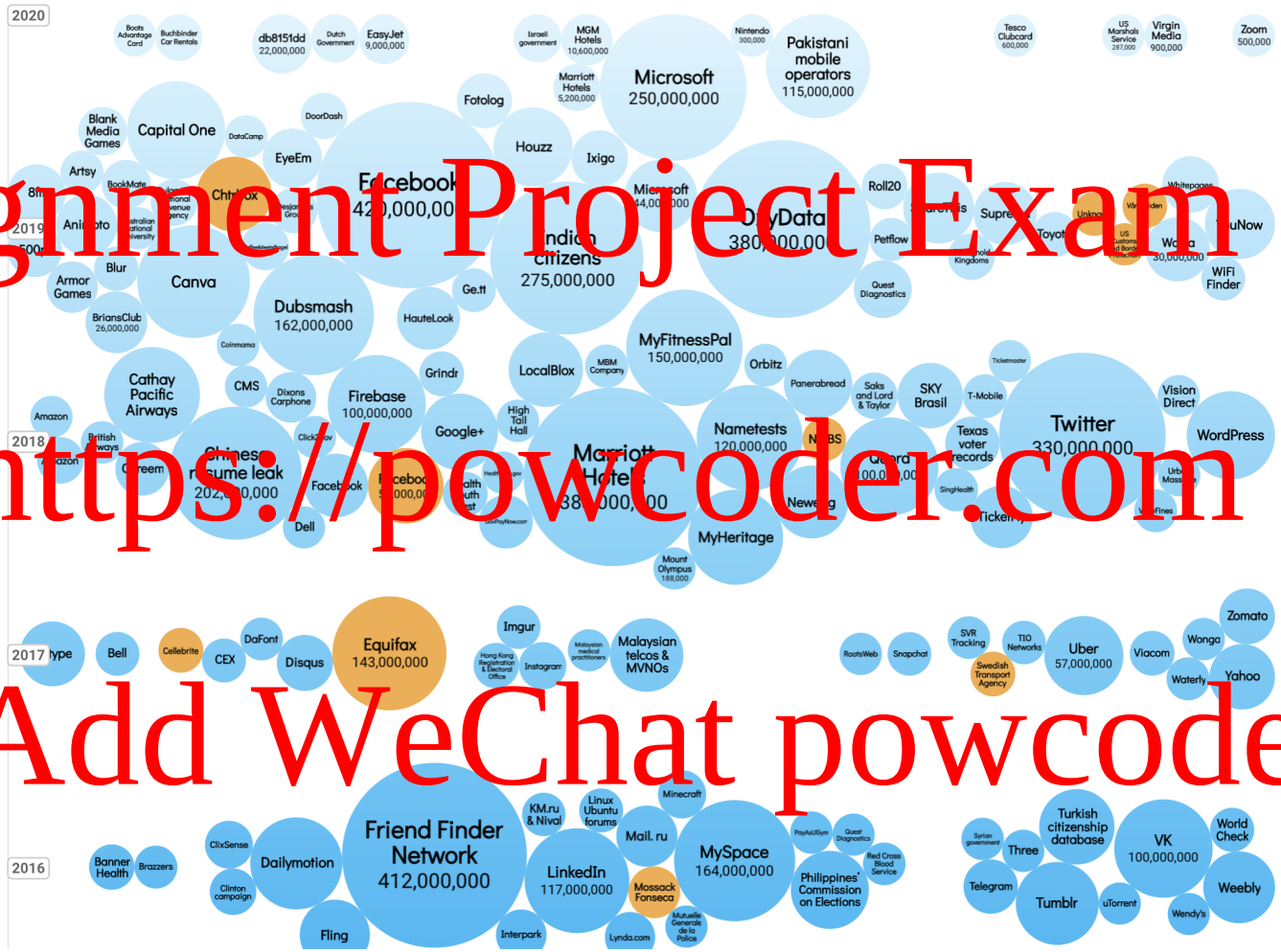


Figure: https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/

Table of Contents

Table of Contents

What is Testability?

- *Testability* refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. A testable system is one that "gives up" its faults easy.
- For a system to be properly testable, it must be possible to control each component's inputs (and possibly manipulate its internal state) and then to observe its outputs. This is often done using a *test harness*.

Why is Testability Important?

- 30-50% of the cost of developing well-engineered systems is taken up by testing [BCK12]
- The later a bug is found, the more costly it is to fix
- Advocates of *Test-Driven Development* believe that by first turning requirements into test cases, and then improving the software to pass the new tests only, results in higher productivity and more modularized, flexible, and extensible code

Common Types of Testing

- *Unit testing* is the testing of an individual unit (module), typically done by a programmer to ensure that the module is behaving as expected
- *Integration testing* is combining a group of components together to verify that they interact correctly together
- *Functional testing* is ensuring that the specified functionality required in the system requirements works
- *Performance testing* is assessing the speed and effectiveness of the system and to make sure it is generating results within a specified time as in performance requirements
- *Stress testing* is evaluating how system behaves under unfavorable conditions (contrast to performance testing)
- *Acceptance testing* is a customer ensuring that the delivered product meets the requirements and works as the customer expected
- *Regression testing* is testing after modification of a system, component, or a group of related units to ensure that the modification is working correctly and is not damaging or imposing other modules to produce unexpected results

Example CI Pipeline

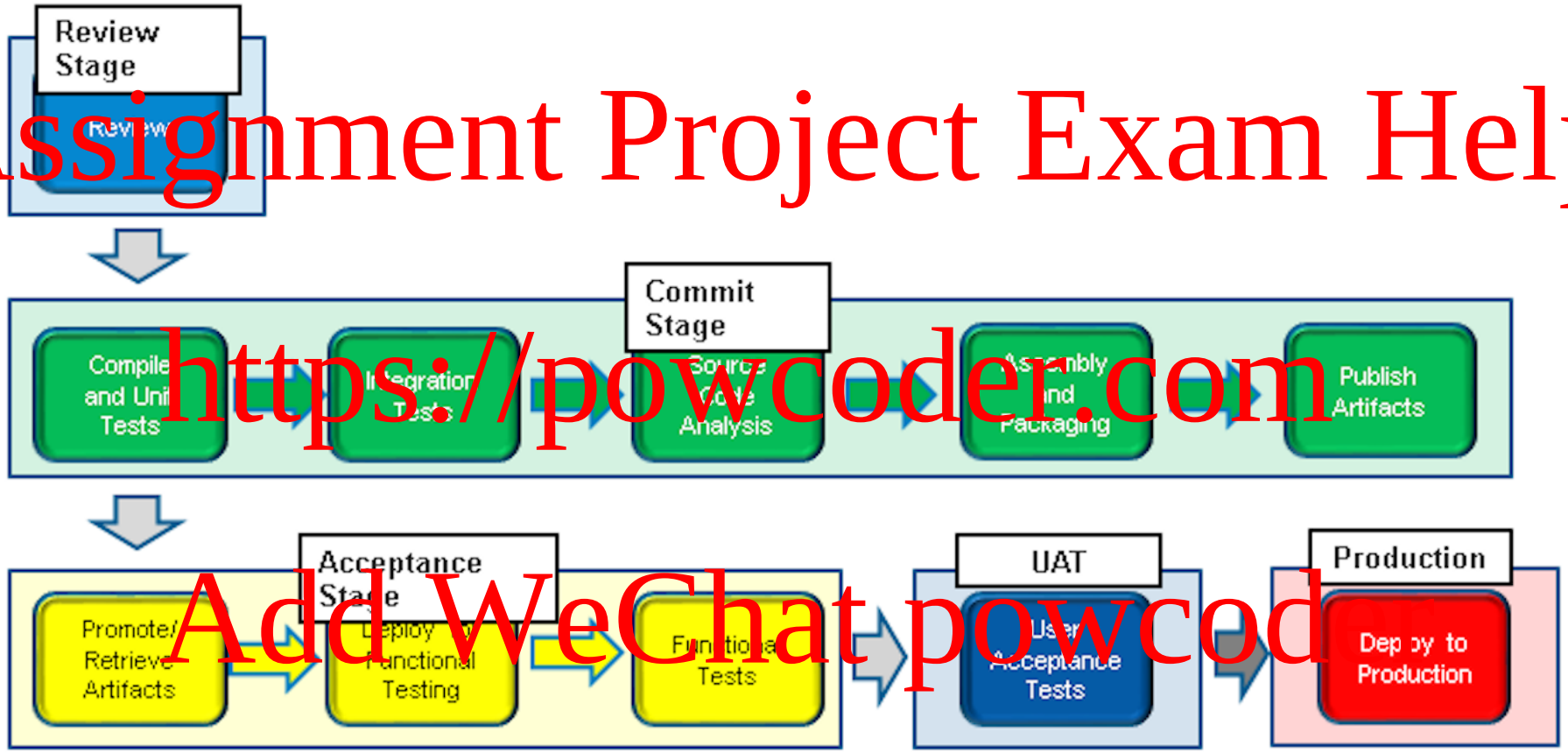


Figure: Example CI Pipeline

Table of Contents

Table of Contents

Specialized Interfaces

- Having *specialized testing interfaces* allows you to control or capture variable values for a component either through a test hardness or through normal execution
- Common examples:
  - A *report* method that returns the full state of an object
  - A *reset* method to set the internal state to a specified internal state
  - A *self-check* method that checks whether all the object's invariants are valid
  - The ability to turn on/off verbose output (very useful to be able to do this as runtime!)
- Be careful not to introduce *security vulnerabilities* here!

Record/Playback

- *Record/playback* refers to both capturing information crossing an interface and using it as input for further testing
- Very useful, but often needs to be built up-front
- In some cases you can find tools to help you with this, e.g. tcpreplay

Localize State Storage

- *Localize state storage* refers to keeping all state for a system/module/etc. in a single place so that it is convenient to start it in an arbitrary state for a test

Abstract Data Sources

- *Abstract data sources* refers to abstracting the interfaces used to access a data source to let you substitute test data more easily
- Commonly implemented with the *repository pattern*
- Closely relates to *mocking* (see later slides)

Sandbox

- *Sandboxing* refers to isolating an instance of the system from the real world to enable experimentation that is unconstrained by the worry about having to undo the consequences of the experiment
- A common technique to implement sandboxing is to *virtualize resources*
  - For example, consider virtualizing the system clock to test Daylight Savings Time boundaries

Executable Assertions

- *Executable assertions* are (usually) hand-coded checks of method pre- and post-conditions and class-level invariants
- When an assertion triggers it means the system is in an *unexpected state*. The challenging question is what to do?
  - Abort? Retry? Ignore? Fail?

Table of Contents

Limit Structural Complexity

- *Limit structural complexity* refers to avoiding or resolving cyclic dependencies between components, isolating and encapsulating dependencies on the external environment, and reducing dependencies between components in general
- The *response* of a class C is a count of the number of methods of C plus the number of methods of other classes that are invoked by the methods of C
  - Keeping this metric low can increase testability
- Consider embracing *eventual consistency* rather than *immediate consistency* to keep your system simpler

Limit Nondeterminism

- *Limit nondeterminism* means to find all sources of nondeterminism, such as unconstrained parallelism, and weeding them out as much as possible
- Nondeterministic systems are extraordinarily hard to test!

Table of Contents

Mocking & Dependency Injection (DI)

- *Mocking*, or using mock objects, refers to using simulated objects that mimic the behavior of real objects in controlled ways
- Mocking is a technique to help make your code easier to unit test, and is commonly seen in test-driven development
- *Mocking frameworks* make creating mock objects quick and easy
- Mocking is usually used with *dependency injection*, which is where one object supplies the dependencies of another object

Mocking & DI Example – Step 1

```
public class FinalInvoiceStep {
  private PrinterServiceImpl printerService = null;
  private EmailServiceImpl emailService = null;

  public FinalInvoiceStep() {
    this.printerService = new PrinterServiceImpl();
    this.emailService = new EmailServiceImpl();
  }

  public void handleInvoice(Invoice invoice, Customer customer) {
    if(customer.prefersEmails()) {
      emailService.sendInvoice(invoice, customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public void testFinalInvoiceStep() {
  FinalInvoiceStep finalInvoiceStep = new FinalInvoiceStep();
  // ???
}
```

- How do I test that the class sends an email if the customer prefers an email?
- How do I test what happens if the printer service fails?

Mocking & DI Example – Step 2

```
public class FinalInvoiceStep {
  private IPrinterService printerService = null;
  private IEmailService emailService = null;

  public FinalInvoiceStep() {
    this.printerService = new PrinterServiceImpl();
    this.emailService = new EmailServiceImpl();
  }

  public void handleInvoice(Invoice invoice, Customer customer) {
    if(customer.prefersEmails()) {
      emailService.sendInvoice(invoice, customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public void testFinalInvoiceStep() {
  FinalInvoiceStep finalInvoiceStep = new FinalInvoiceStep();
  // ???
}
```

- Have the invoice step use interfaces rather than implementations for the printer and email service
- Now I can theoretically use a test printer service, but how do I get the class to use it?

Mocking & DI Example – Step 3

```
public class FinalInvoiceStep {
  private IPrinterService printerService = null;
  private IEmailService emailService = null;

  public FinalInvoiceStep(IPrinterService printerService,
                          IEmailService emailService) {
    this.printerService = printerService;
    this.emailService = emailService;
  }

  public void handleInvoice(Invoice invoice, Customer customer) {
    if(customer.prefersEmails()) {
      emailService.sendInvoice(invoice, customer.getEmail());
    } else {
      printerService.printInvoice(invoice);
    }
  }
}

public class TestPrinterService implements IPrinterService { ... }
public class TestEmailService implements IEmailService { ... }

public void testFinalInvoiceStep() {
  IPrinterService printerService = new TestPrinterService();
  IEmailService emailService = new TestEmailService();
  FinalInvoiceStep finalInvoiceStep =
      new FinalInvoiceStep(printerService, emailService);
  // TestPrinterService can be told to simulate a failure
}
```

- Have FinalInvoiceStep require the printer and email service in the constructor
- This is dependency injection
- Writing all these test classes is a real pain!

Mocking & DI Example – Step 4


```
public void testFinalInvoiceStep() {
    IPrinterService printerService =
        Mockito.mock(IPrinterService.class);
    Mockito.when(printerService.printInvoice())
        .thenThrow(new PrinterOutOfServiceException());
    // ...

    FinalInvoiceStep finalInvoiceStep =
        new FinalInvoiceStep(printerService, emailService);

    finalInvoiceStep.handleInvoice(...);
    // Test to make sure the invoice class
    // behaves as expected when the printer
    // has failed
}
```


- Mocking frameworks make writing test objects easy!

Architecting for Testability

- Keep testability in mind when designing your components
- If your component is hard to test in isolation, that may be a good sign that it has too many responsibilities and should be decomposed
- Your QA team is excellent at testing functionality, but how are they at testing non-functional requirements? Will you need to give them training, tooling?
- There are a ton of tools out there to help you out. For example, coworker of mine reported great success in using clumsy to simulate network outage scenarios. Google is your friend!
- Some things are remarkably hard to test in non production environments. For example, Facebook would be hard-pressed to simulate production levels of load on non-production servers. If you build great deployment strategies (blue-green, rollback, etc.) test-in-production might become a viable strategy. If so, then consider using:
  - *A/B testing* (a.k.a. *split testing*) is essentially an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal.
  - *Canary testing* is pushing changes to a small group of end users who are unaware that they are receiving new code

Netflix's Simian Army – Aggressively Testing at Runtime

- *Chaos Monkey* – identifies groups of systems and randomly terminates one of the systems in a group
- *Latency Monkey* – introduces artificial delays in the client-server communication layer to simulate service degradation and measures if upstream services respond appropriately
- *Conformity Monkey* – finds instances that don't adhere to best practices and shuts them down
- *Doctor Monkey* – taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances
- *Janitor Monkey* – searches for unused resources and disposes of them
- *Security Monkey* – finds security vulnerabilities or violations and terminates the offending instances
- *10-18 Monkey* – detects configuration and runtime problems in instances serving customers in multiple geographic regions
- *Chaos Gorilla* – simulates the outage of an entire Amazon availability zone

Optional Reading

- SparkJoy by Running Fewer Tests
- terraform-compliance
- Automated Testing for Terraform, Docker, Packer, Kubernetes, and More
- Microservices Testing Strategies, Types & Tools: A Complete Guide
- Building in compliance in your CI/CD pipeline with conftest
- Testing in Production: the hard parts
- Testing Memory Allocators: ptmalloc2 vs tcmalloc vs hoard vs jemalloc While Trying to Simulate Real-World Loads

Table of Contents

Table of Contents

What is Interoperability?

- *Interoperability* is the degree to which two or more systems can usefully exchange meaningful information via interfaces in a particular context
- Includes not only the ability to exchange data (*syntactic interoperability*) but also the ability to correctly interpret the data being exchanged (*semantic interoperability*)

Why is Interoperability Important?

- Interoperability is what allows different systems to communicate and work together. It is the foundational idea of the Internet.
- Interoperability allows you to construct capabilities from existing systems, e.g. embed a Google Maps widget inside your mobile app
- Interoperability allows you to provide a service to used by unknown systems in the future

How to Achieve Interoperability?

- Interoperability is achieved by having systems speak a common protocol. It is most frequently achieved by applying techniques from *service-oriented architecture (SOA)*.
- An *interface* is the set of assumptions that you can safely make about an entity. Interfaces are the key to interoperability.
- Interfaces in this context is beyond just an API (*syntax*): it includes the meaning and expectations behind the system.

Service-Oriented Architecture

- A *service* is a collection of self-contained business functionality that exist as applications with their own design characteristics that support the strategic goals of the organization.
- *Service-orientation* is design paradigm comprised of a set of design principles that are applied to application logic that is represented as services.
- *Service-oriented architecture (SOA)* is an architectural style that supports service orientation.

- *Discovery* refers to allowing a consumer of a service to discover the location, identity, and interface of a service
- *Handling of the response*, which may be:
  - The service reports back to the requester with the response
  - The service sends its response on to another system
  - The service broadcasts its response to any interested parties

Table of Contents

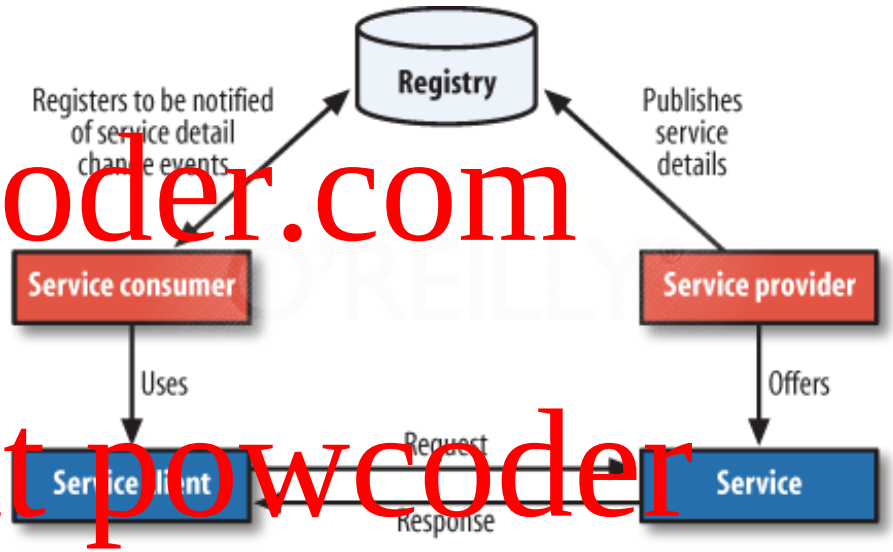
Table of Contents

Discover Service

- *Discover service* means to locate a service through searching a known directory service.
- By *service*, we simply mean a set of capabilities that is accessible via some kind of interface.
- The basic problem that discover service is trying to solve is "How does the client of a service make requests to a dynamically changing set of ephemeral service instances?"

Discover Service – Service Registry

- Problem: How do clients of a service and/or routers know about the available instances of a service?
- Solution: Use a service registry, which is a database of services, their instances, and their locations. Service instances are registered with the service registry and deregistered on shutdown.
- Benefits: Discovery is enabled.
- Drawbacks: The service registry is a critical system component and must be highly available.
- Example service registries: Netflix Eureka, Apache Zookeeper, Consul, Etcd, and many implicitly-provided ones



Service Registry Example

```
http1.json:
{
  "ID": "http1",
  "Name": "http",
  "Address": "172.17.0.3",
  "Port": 80,
  "check": {
    "http": "http://172.17.0.3:80",
    "interval": "10s",
    "timeout": "1s"
  }
}
$ curl -X PUT --data-binary @http1.json \
    http://.../v1/agent/service/register
```

Listing 1: Registering a service with Consul

```
$ curl -s http://.../v1/catalog/service/http | jq .
[
  {
    "ModifyIndex": 372,
    "CreateIndex": 372,
    "Node": "myconsul",
    "Address": "172.17.0.2",
    "ServiceID": "http1",
    "ServiceName": "http",
    "ServiceTags": [],
    "ServiceAddress": "172.17.0.3",
    "ServicePort": 80,
    "ServiceEnableTagOverride": false
  }
]
```

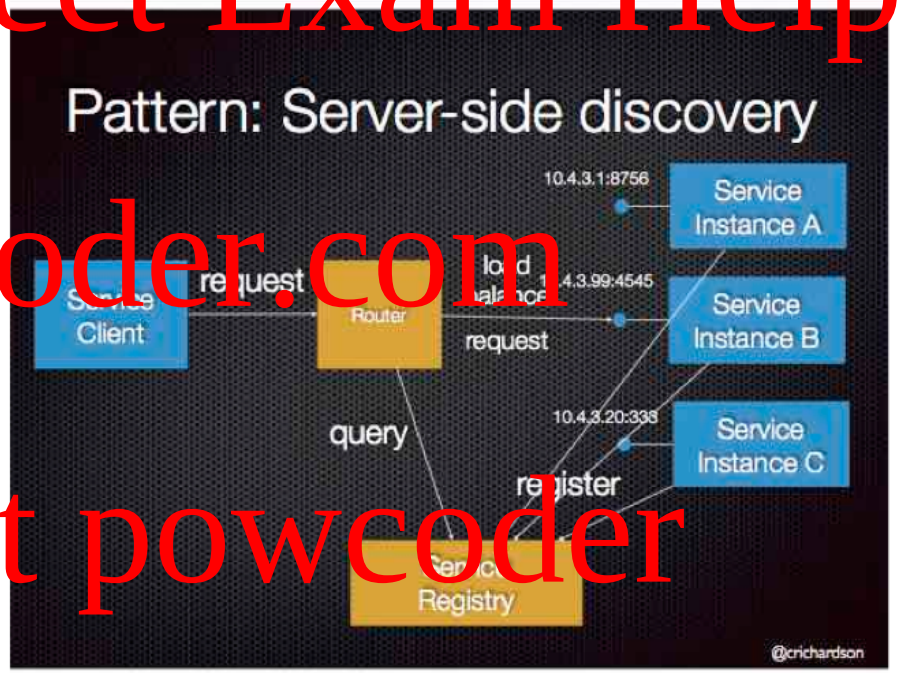
Listing 2: Discovering services with Consul

Consul [1] is an open source service registry.

[1] https://www.consul.io/

Discover Service – Server-Side Discovery

- Server-side discovery: The client makes a request via a router (a.k.a. load balancer) that runs at a well-known location. The router queries the service registry to find available service instances.
- Benefits: Simpler client code; capability often provided by environment
- Drawbacks: Router is a critical system component and must be highly available, more network hops required
- Example server-side discovery technologies: AWS ELB, F5 BigIP, HAProxy



Discover Service – Client-Side Discovery

- Client-side discovery: The client queries the service registry and makes direct connections to the service instances
- Benefits: Fewer moving parts and network hops compared to server-side discovery
- Drawbacks: Must reimplement discovery logic in each programming language/framework used by your application; clients must handle service instance failure; service instances are much harder to firewall and isolate, and frequently cannot be directly accessed over the Internet; how do you do clean shutdowns and connection draining for your service?

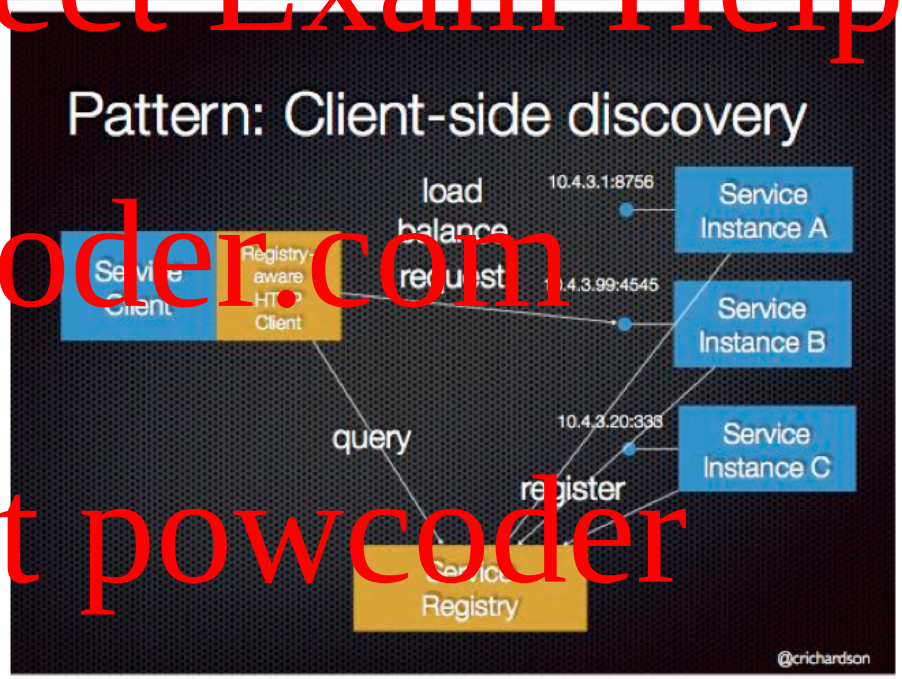
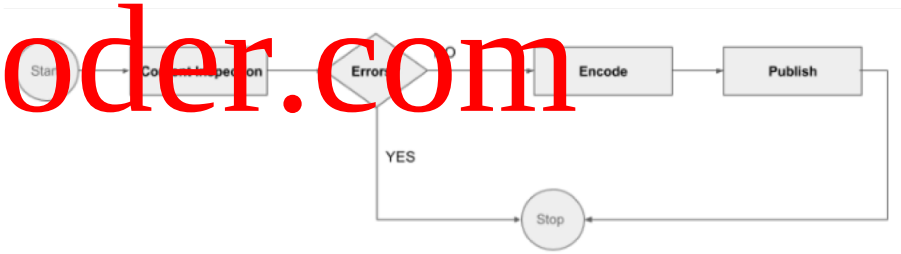


Table of Contents

Orchestrate

- To *orchestrate* means to use a control mechanism to coordinate and manage and sequence the invocation of particular services (which could be ignorant of each other). It is about mapping out workflow.
- Conceptually identical to the *mediator* design pattern
- Example technologies: BPEL, Ansible, Netflix Conductor, Mule



Tailor Interface

- To *tailor interface* is to add or remove capabilities to an interface, e.g. translation, buffering, or smoothing
- Conceptually identical to the *decorator* design pattern

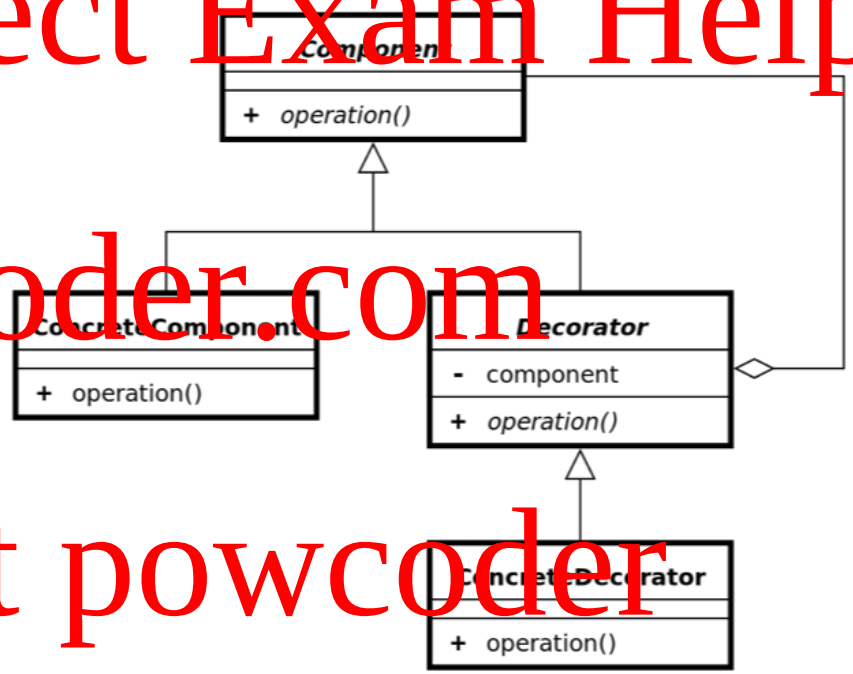
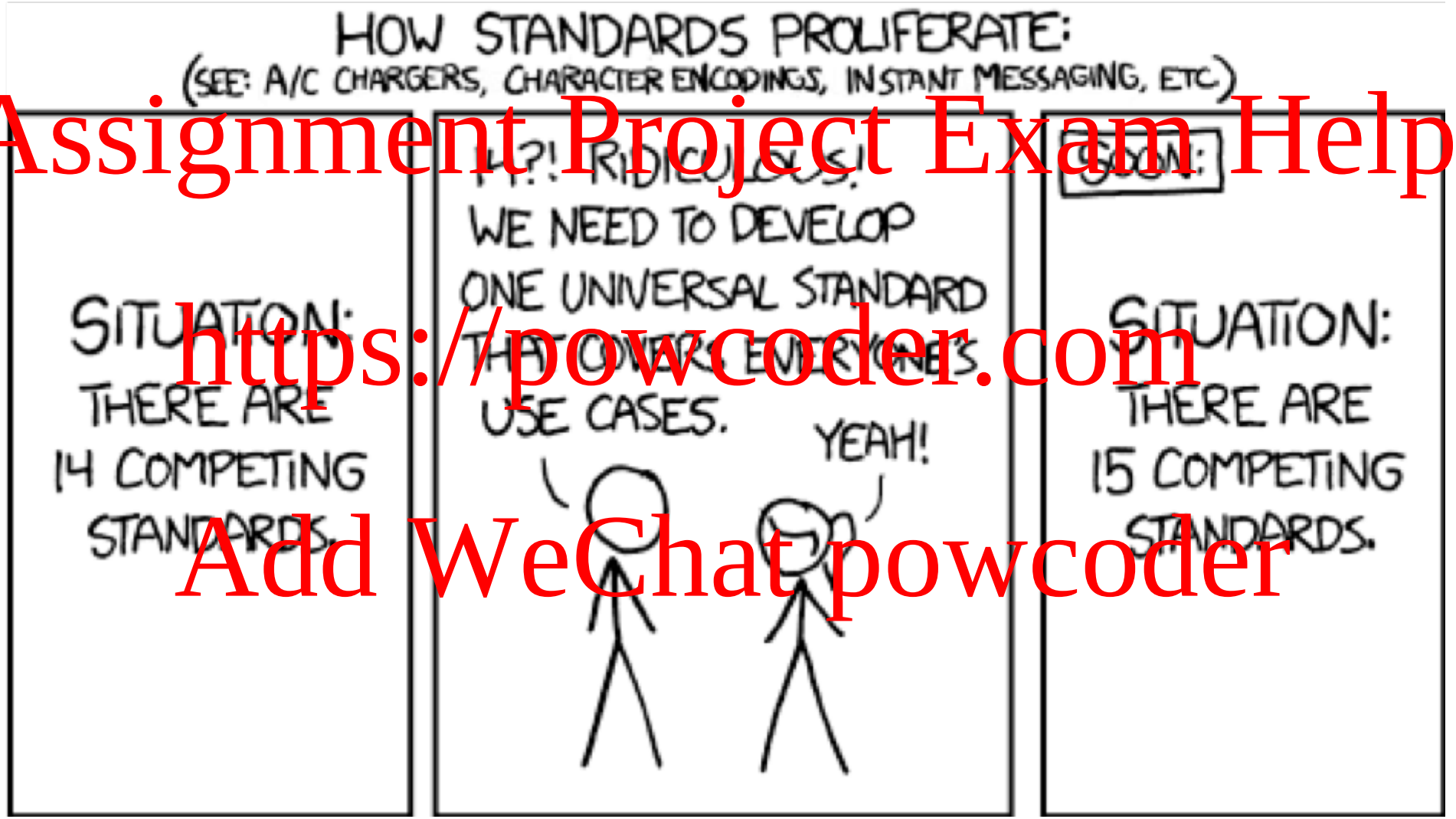


Table of Contents

Standards and Interoperability

- Notable standards bodies: IEEE, IETF, ISO, W3C
- Standards are *not enough* to guarantee interoperability
- Standards are incredibly numerous, inconsistent, open-ended, evolve over time, are of varying quality, and may be used as "weapons" by competing organizations
- Implementations are similarly horrible. Here's an entire RFC on observed implementation problems with TCP: https://tools.ietf.org/html/rfc2525

Standards and Interoperability



Standards and Interoperability

- HTML, CSS, and JavaScript are all standardized
  - HTML 2.0 [November 1995], 3.2, 4.0, 5, 5.1 [November 2016]
  - CSS 1 [December 1996], 2, 2.1, 3 [ongoing]
  - ECMAScript 1 [June 1997], 2, ..., 8 [June 2017]
- Is it interoperable?
- Do you write to the standard or to a particular implementation?

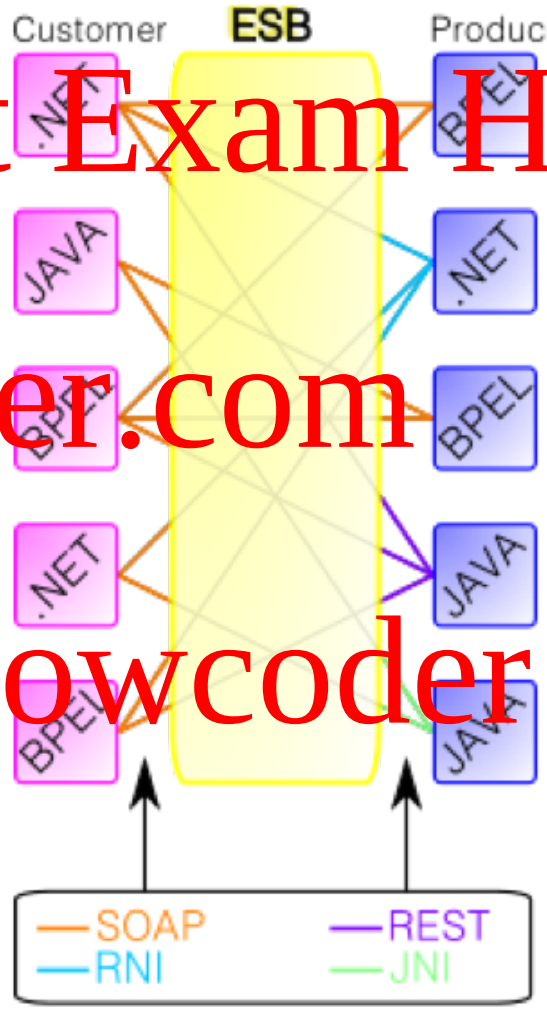
Table of Contents

SOA Protocols

- The two most common web-based interoperation protocols are SOAP and REST
- Both typically communicate over HTTP
- REST tends to be simpler, SOAP tends to be more "complete"
- REST seems more commonly used nowadays, but these technologies are very trendy. Remember DCOM? COBRA? XML-RPC? Perhaps GraphQL or another protocol will soon replace REST.
- When efficiency is paramount, architects sometimes choose cross-platform binary protocols like Apache Thrift

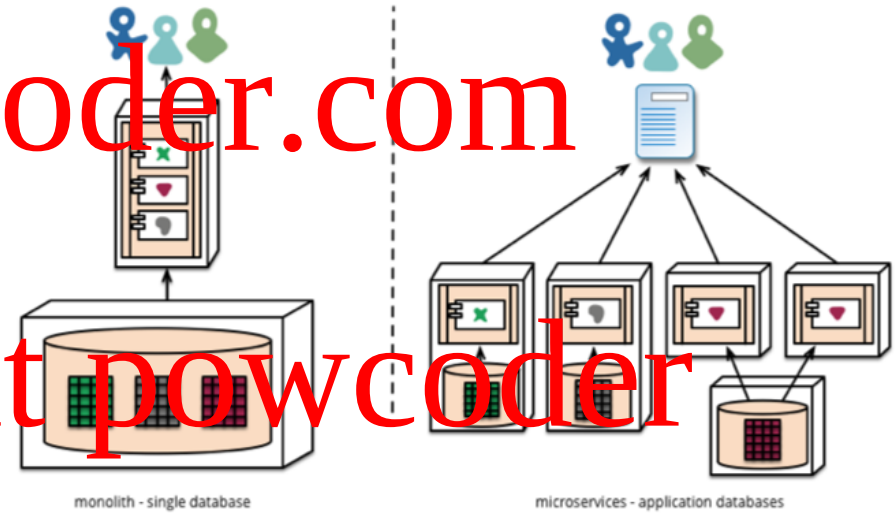
Enterprise Service Bus

- An enterprise service bus (ESB) implements a communication system between mutually interacting software applications in a service-oriented architecture (SOA).
- The primary goal of the high-level protocol communication is enterprise application integration (EAI) of heterogeneous and complex service or application landscapes (a view from the network level).
- Includes invocation, routing, mediation, messaging, orchestration, security, etc.



Microservices

- The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating via lightweight mechanisms, often a HTTP resource API
- Principles
  - Built around business capabilities
  - Independently deployable
  - Smart endpoints and dumb pipes (the opposite of the ESB)
  - Product, not project
  - Highly decentralized (data management, governance, programming language, etc.)



Architecting for Interoperability

- Embrace standards where you can, as it allows you to benefit from existing code, libraries, tools, etc.
- Ensure your interfaces are clear, well-defined, and do not leak implementation details
- When designing a service, be sure to consider:
  - RPC vs. Message vs. Resource APIs
  - Synchronous vs. asynchronous operations
  - Transactions
  - Idempotence
  - Backwards compatibility
  - Versioning
  - *Postel's law*: Be conservative in what you send, be liberal in what you accept

Optional Readings

- API design: Understanding gRPC, OpenAPI and REST and when to use them

Table of Contents

Table of Contents

What is Performance?

- *Performance* is about characterizing the events that can occur (and when they can occur) and the system or element's time-based response to these events
- All systems have *performance requirements*, even if they aren't explicitly expressed

Why is Performance Important?

- Adding a 1-second delay to the Bing search results response time decreased user satisfaction by 1.6% and decreased revenue per user by 2.3% [Chr10]
  - Bing's annual revenue is approximately $1 billion, so this is a $23 million/year decrease

Why is Performance Important?

- Adding a 1-second delay to the Bing search results response time decreased user satisfaction by 1.6% and decreased revenue per user by 2.3% [Chr10]
  - Bing's annual revenue is approximately $1 billion, so this is a $23 million/year decrease
- When Edmunds reduced load times from 9 seconds to 1.4 seconds, ad revenue increased 3% and page views per session went up 17% [Ros12]

Why is Performance Important?

- Adding a 1-second delay to the Bing search results response time decreased user satisfaction by 1.6% and decreased revenue per user by 2.3% [Chr10]
  - Bing's annual revenue is approximately $1 billion, so this is a $23 million/year decrease
- When Edmunds reduced load times from 9 seconds to 1.4 seconds, ad revenue increased 3% and page views per session went up 17% [Ros12]
- When Shopzilla dropped latency from 7 seconds to 2, revenue went up 7-12%, page views increased 25%, and hardware costs dropped 50% [Dix09]

Why is Performance Important?

- Adding a 1-second delay to the Bing search results response time decreased user satisfaction by 1.6% and decreased revenue per user by 2.3% [Chr10]
  - Bing's annual revenue is approximately $1 billion, so this is a $23 million/year decrease
- When Edmunds reduced load times from 9 seconds to 1.4 seconds, ad revenue increased 3% and page views per session went up 17% [Ros12]
- When Shopzilla dropped latency from 7 seconds to 2, revenue went up 7-12%, page views increased 25%, and hardware costs dropped 50% [Dix09]
- In 2012, Amazon estimated a 1 second page load slowdown would cost it $1.6 billion in sales per year [Eat12]

Why is Performance Important?

- Adding a 1-second delay to the Bing search results response time decreased user satisfaction by 1.6% and decreased revenue per user by 2.3% [Chr10]
  - Bing's annual revenue is approximately $1 billion, so this is a $23 million/year decrease
- When Edmunds reduced load times from 9 seconds to 1.4 seconds, ad revenue increased 3% and page views per session went up 17% [Ros12]
- When Shopzilla dropped latency from 7 seconds to 2, revenue went up 7-12%, page views increased 25%, and hardware costs dropped 50% [Dix09]
- In 2012, Amazon estimated a 1 second page load slowdown would cost it $1.6 billion in sales per year [Eat12]
- The less efficient the application, the more *battery power* it will consume

Why is Performance Important?

- Adding a 1-second delay to the Bing search results response time decreased user satisfaction by 1.6% and decreased revenue per user by 2.3% [Chr10]
  - Bing's annual revenue is approximately $1 billion, so this is a $23 million/year decrease
- When Edmunds reduced load times from 9 seconds to 1.4 seconds, ad revenue increased 3% and page views per session went up 17% [Ros12]
- When Shopzilla dropped latency from 7 seconds to 2, revenue went up 7-12%, page views increased 25%, and hardware costs dropped 50% [Dix09]
- In 2012, Amazon estimated a 1 second page load slowdown would cost it $1.6 billion in sales per year [Eat12]
- The less efficient the application, the more *battery power* it will consume
- Bitcoin mining is a race, and we're basically at the point where only specially-designed hardware is economically competitive
  - Bitcoin mining uses an incredible amount of electricity!
  - If you create a significantly faster Bitcoin mining system, you can essentially print money

Perceived Performance

- *Perceived performance* refers to how quickly a software feature appears to perform its task
- Startup screens, spinners, etc. make your application marginally slower, but make it *feel faster* to humans
- Never forget the *human element* in software architecture

Event Types

- *Periodic* events arrive predictably at regular time intervals
- *Stochastic* events arrive according to some probabilistic distribution
- *Sporadic* events arrive according to a pattern that is nearly periodic or stochastic

Performance Response Measures

- *Latency* is the time between the arrival of the stimulus and the system's response to it
- *Deadlines in processing* are the point in time at which the stimulus must be processed (the system cannot be late)
  - More common in real-time systems, e.g., a engine controller which powers fuel ignition
- *Throughput* is the number of transactions the system can process in a unit of time
- *Jitter* is the allowable variation in latency
- *Miss rate* is the number of events not processed because the system was too busy to respond

Rule of 10

- 0.1 seconds (100ms) is about the limit for having the user feel that the system is *reacting instantaneously*
- 1 second is about the limit for the *user's flow of thought to stay uninterrupted*, even though the user will notice the delay
- 10 seconds is about the limit for *keeping the user's attention* focused on the dialogue

Latency Numbers Every Programmer Should Know

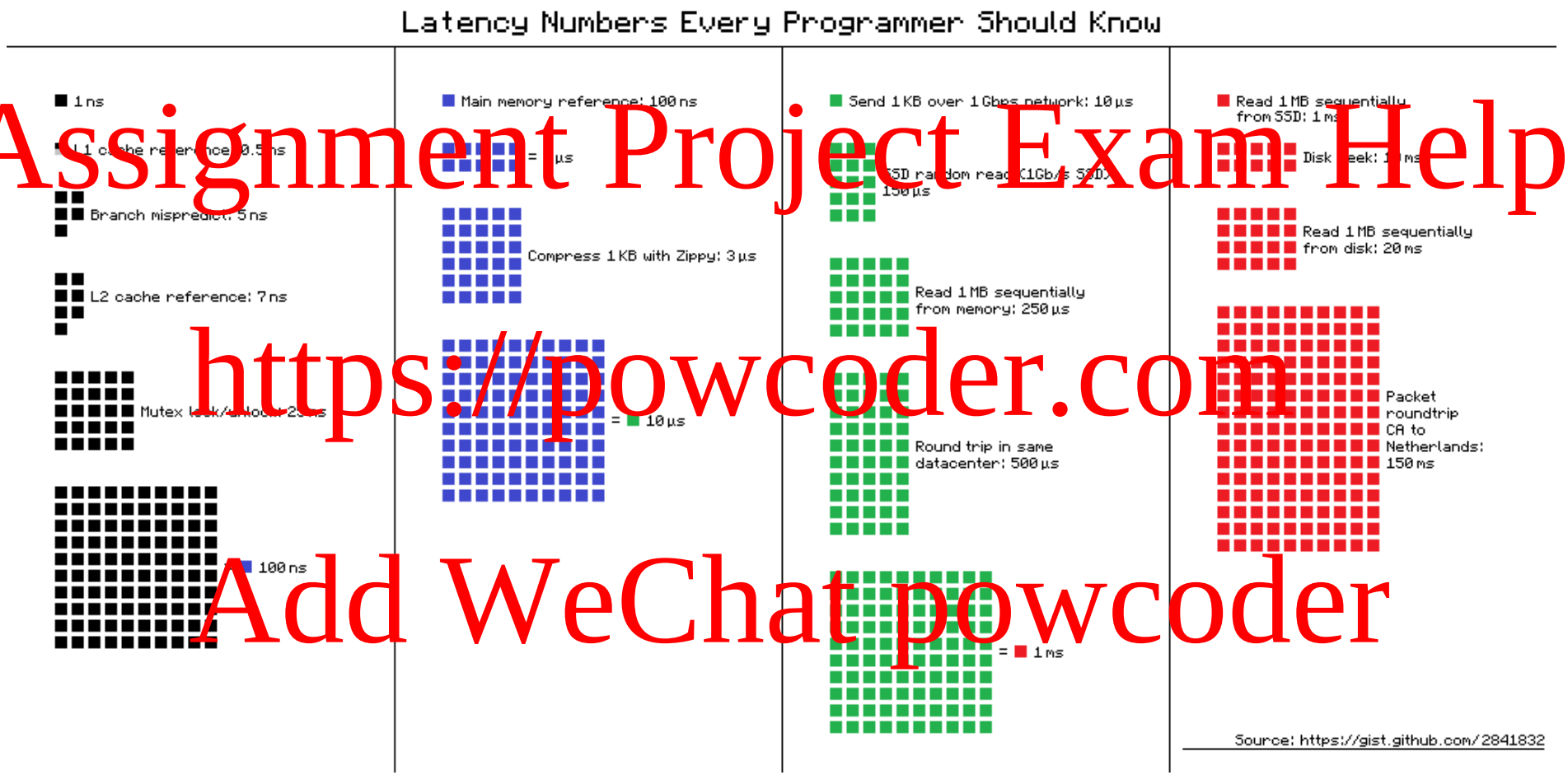


Figure: Latency Numbers Every Programmer Should Know [Bon12]

Contributors to Response Time

- *Processing time* is when the system is working to respond
- *Blocked time* is when the system is unable to respond
- *Causes of blocked time*
  - *Contention for resources* – clients must wait for access to a resource
  - *Availability of resources* – the resource might be offline due to failure
  - *Dependency on other computation* == this computation is dependent on the result of another, in-progress computation
- *Asynchrony* is largely about eliminating blocked time by allowing the computer to process other work while it is waiting
- Your computer has lots of *mostly-hidden blocked time* in it
  - CPU failed branch prediction (more on this later)
  - Context (thread) switches
  - Page faults

Table of Contents

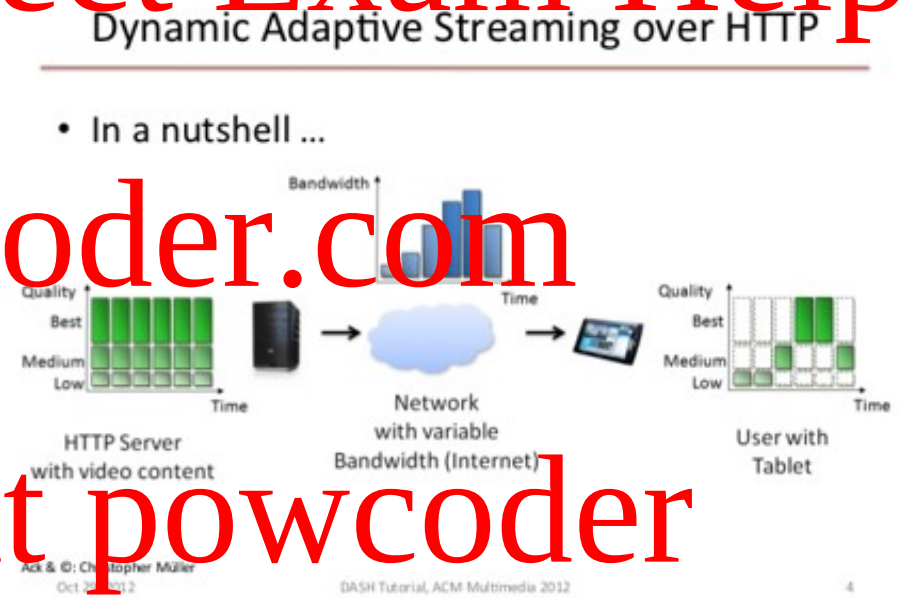
Performance Tactic Categories

- *Control Resource Demand* – carefully manage the demand for resources and that the resources you have are applied judiciously
- *Manage Resources* – make resources at hand work more effectively in handling the demands put to them

Table of Contents

Manage Sampling Rate

- *Reducing the sampling frequency* at which a stream of environmental data is captured, with a corresponding loss of fidelity
- Analogous to YouTube's adaptive bitrate streaming



Limit Event Response

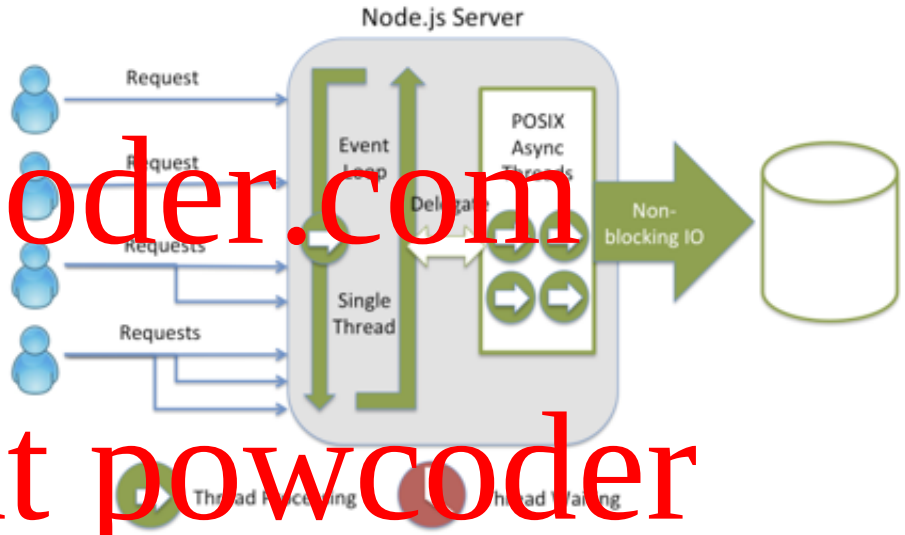
- Only process events up to a set maximum rate, ensuring more predictable processing when the events are actually processed, and queue events that you cannot process in time
- Humans often prefer slightly-higher average response time with low jitter over slightly-lower average response with high jitter
- What do you do if your queue is insufficient to handle the worst case? Drop old events? Ignore new events? (*Backpressure* is covered in *Manage Resources*)

Prioritize Events

- Rank events according to how important it is to service them
- Process events in priority order
- Consider ignoring low-priority events
- Related to scheduling (later)

Reduce Overhead

- *Remove intermediaries* to improve latency (when we talk about modifiability, we will talk about *adding intermediaries*)
- *Co-locate resources* (next slide)
- Perform *periodic cleanup* of resources that have become inefficient
- Use *single-threaded event-loop processor design* (like Node.JS) to avoid contention



Co-Locate Resources

- Avoid time delay of communication by putting processing & data in the same place
- Big Data maxim: *move computation to data, not data to computation*
  - Fundamental to Hadoop, but applies in many circumstances

- My most commonly-used performance improvement "trick"!
  - The vast majority of performance problems I've experienced have been I/O bound, not CPU bound

- In the cloud, networking has become so advanced that this adage may no longer apply

Co-Locate Resources Example 1

- SQL database with a table of 10,000,000 records, where each record is 1K, being accessed by a cluster of web servers.
- Objective: Implement a table that points to this table and supports sorting, filtering, and pagination
- *Question*: Should we perform the sorting, filtering, and pagination in the database, the web server, or the client side (i.e., in the web browser)? What are the tradeoffs involved?
- Useful figures:
  - Average connection speed in the United States as of Q3 2015: 12.6 megabits/second (1.6MB/sec)
  - AWS m4.large: 2CPU, 8GB RAM, $0.10/hr
  - AWS x1.32xlarge: 128CPU, 1,952TB RAM, $13.338/hr
  - Network: 10 gigabits/second (1.25 GB/sec)
  - SSD read throughput: 550MB/sec per drive
  - SATA 3.2 interface limit: 16 gigabits/second (2 GB/sec)
  - DDR2-800 RAM bandwidth: 12.8GB/sec

Co-Locate Resources Example 2

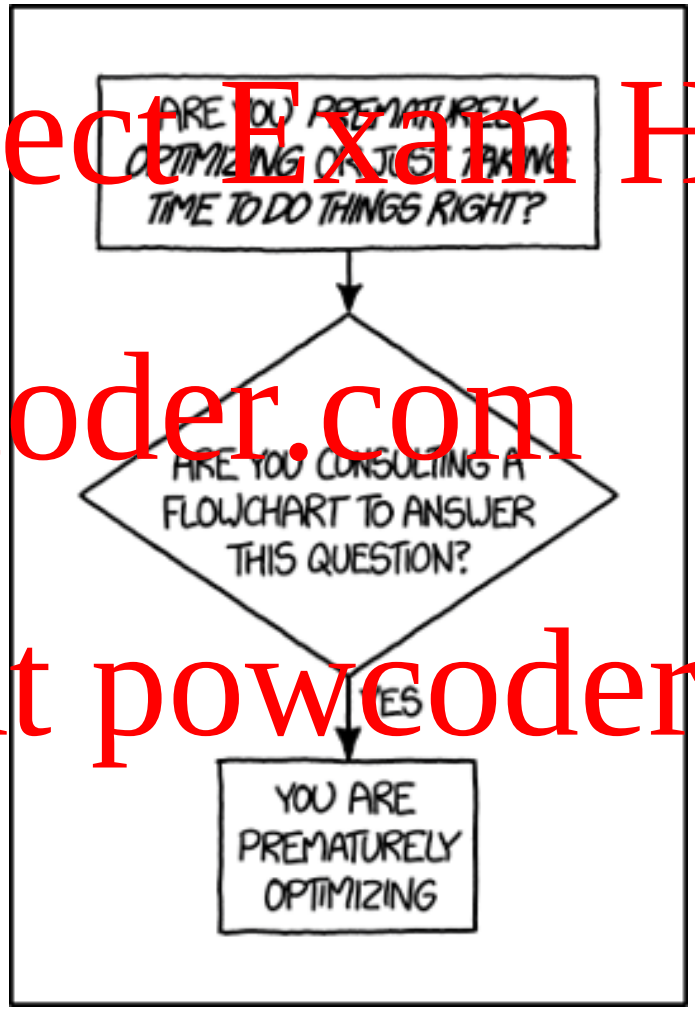
- SQL database with a table of 10,000,000 records, where each record is 1K, being accessed by a cluster of web servers.
- Objective: Calculate statistics (e.g. average, mean, median) for the numerical columns in the table
- *Question*: Should we calculate these statistics in the database, the web server, or the client side (i.e. in the web browser)? What are the tradeoffs involved?
- Useful figures:
  - Average connection speed in the United States as of Q3 2015: 12.6 megabits/second (1.6MB/sec)
  - AWS m4.large: 2CPU, 8GB RAM, $0.10/hr
  - AWS x1.32xlarge: 128CPU, 1,952TB RAM, $13.338/hr
  - Network: 10 gigabits/second (1.25 GB/sec)
  - SSD read throughput: 550MB/sec per drive
  - SATA 3.2 interface limit: 16 gigabits/second (2 GB/sec)
  - DDR2-800 RAM bandwidth: 12.8GB/sec

Bound Execution Times

- Place a limit on how much execution time is used to respond to an event
- Most useful for iterative, data-dependent algorithms, where you can tradeoff between time & accuracy, otherwise this is more of an availability tactic
  - Monte Carlo # of simulations

Increase Resource Efficiency

- Improve the algorithms used in critical areas
- Donald Knuth: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%" [Knu74]
- *Always* use tools (e.g. a *profiler*) and a data-driven approach to find that 3%
  - Many performance problems are due to non-obvious causes
  - Your intuition, and mine, for performance hotspots is terrible



Increase Resource Efficiency Non-Obvious Example

```
#include ...

void do_work()
{
    // Generate data
    const unsigned arraySize = 32768;
    int data[arraySize];
    for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;

    // !!! With this, the next loop runs 6 times faster !!!
    std::sort(data, data + arraySize);

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i)
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime =
      static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    std::cout << elapsedTime << std::endl;
    std::cout << "sum = " << sum << std::endl;
}
```

Increase Resource Efficiency Non-Obvious Example

```
#include ...

void do_work()
{
    // Generate data
    const unsigned arraySize = 32768;
    int data[arraySize];
    for (unsigned c = 0; c < arraySize; ++c)
        data[c] = std::rand() % 256;

    // !!! With this, the next loop runs 6 times faster !!!
    std::sort(data, data + arraySize);

    // Test
    clock_t start = clock();
    long long sum = 0;
    for (unsigned i = 0; i < 100000; ++i)
    {
        // Primary loop
        for (unsigned c = 0; c < arraySize; ++c)
        {
            if (data[c] >= 128)
                sum += data[c];
        }
    }

    double elapsedTime =
      static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
    std::cout << elapsedTime << std::endl;
    std::cout << "sum = " << sum << std::endl;
}
```


Table of Contents

Increase Resources

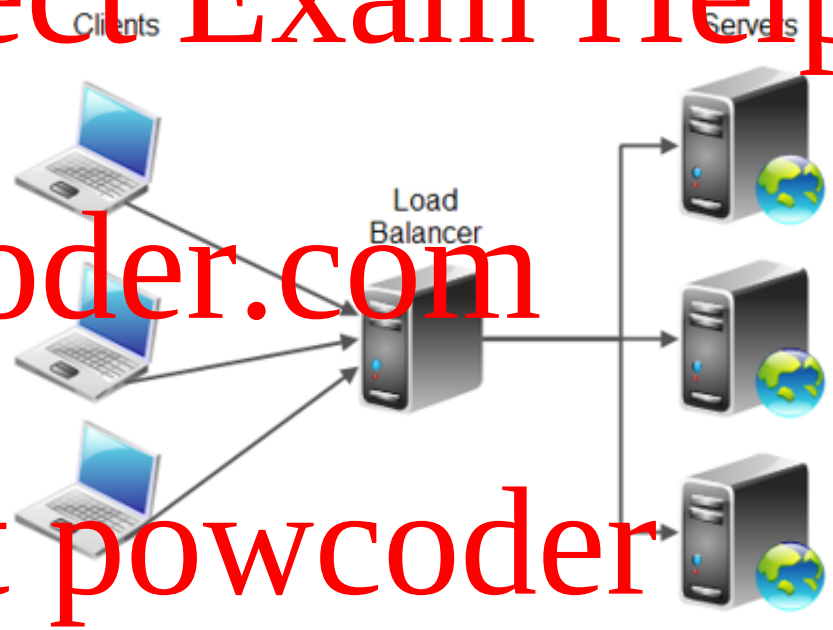
- Buy your way out of the problem!
- Use faster processors, additional processors, additional memory, etc.
- In many cases, the *cheapest and fastest way* to make an improvement

	SSD	HDD
Capacity	1TB	1TB
IOPS (4K read)	9,488	45
Throughput (sequential read)	500MB/s	143MB/s
Seek time	0.0342ms	8.88ms
Cost (2017)	$179.99	$64.99

- Process more requests in parallel to reduce blocking time
- Use *scheduling policies* to achieve goals you find desirable (see later)

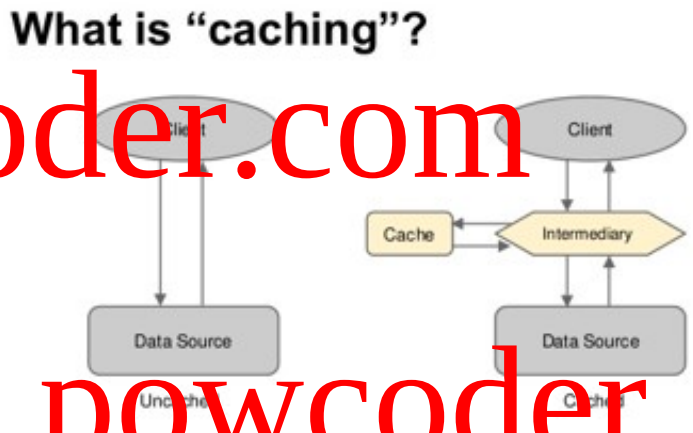
Maintain Multiple Copies of Computations

- Introduce replicas to reduce the contention that would occur if all computations took place on a single server
- Often paired with a *load balancer* to assign new work to one of the available duplicate servers



Maintain Multiple Copies of Data

- *Caching* is about keeping copies of data (possibly one a subset of the other) on storage with different access speeds (more on next slide)
- *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses
- Don't underestimate the challenge required to keep data copies consistent and synchronized



Caching Design Considerations

- How many? What sizes?
- Distributed? Shared?
- Replacement policy
  - FIFO, LIFO, LRU, etc.
- Expiration policy
  - Absolute, sliding, etc.
- Write policy
  - *Write-through caching* directs write I/O onto cache and through to underlying permanent storage before confirming I/O completion to the host.
  - *Write-back caching* directs write I/O cache and completion is immediately confirmed to the host. The cache asynchronously copies the data to the permanent storage at a later time.
  - *Write-around caching* directs write I/O directly to permanent storage, bypassing the cache.

Bound Queue Sizes

- Limit the number of queued arrivals and consequently the resources used to process the arrivals
- Can you prevent clients from enqueueing new data (i.e. implement *backpressure*)? If not, how will you handle clients exceeding your queue limits?
  - Remember Limit Event Response?

Schedule Resources

- When there is contention for a resource, the resource must be *scheduled*.
- To schedule you must choose a *scheduling policy*.

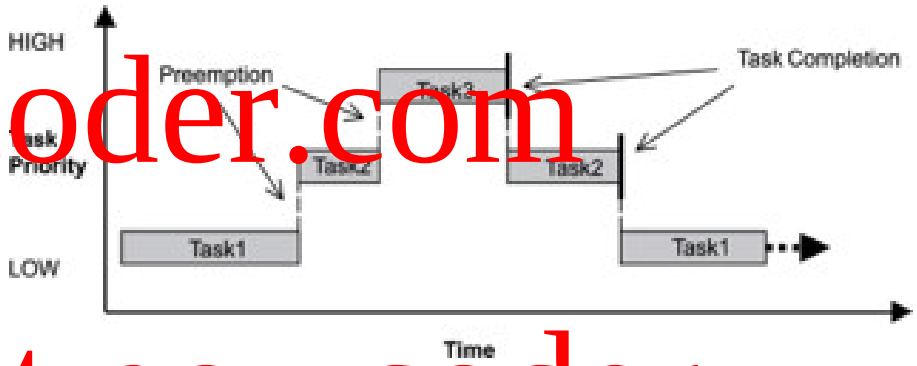
Table of Contents

Scheduling Overview

- A *scheduling policy* is the strategy by which a *scheduler* decides which work should be executed next
- A scheduling policy conceptually has two parts:
  - Priority assignment
  - Dispatching
- *Competing criteria* for scheduling
  - Optimal resource usage
  - Request importance
  - Minimizing the number of resources used
  - Minimizing latency
  - Maximizing throughput
  - Preventing starvation to ensure fairness

Scheduling Concepts

- *Preemption* – can a task be temporarily interrupted with the intention of resuming the task at a later time?
- *Priority inversion* – a high priority task is indirectly preempted by a lower priority task
- *Priority boosting* – can lower priority tasks have their temporarily priority boosted to avoid deadlock?



Priority Inversion Example

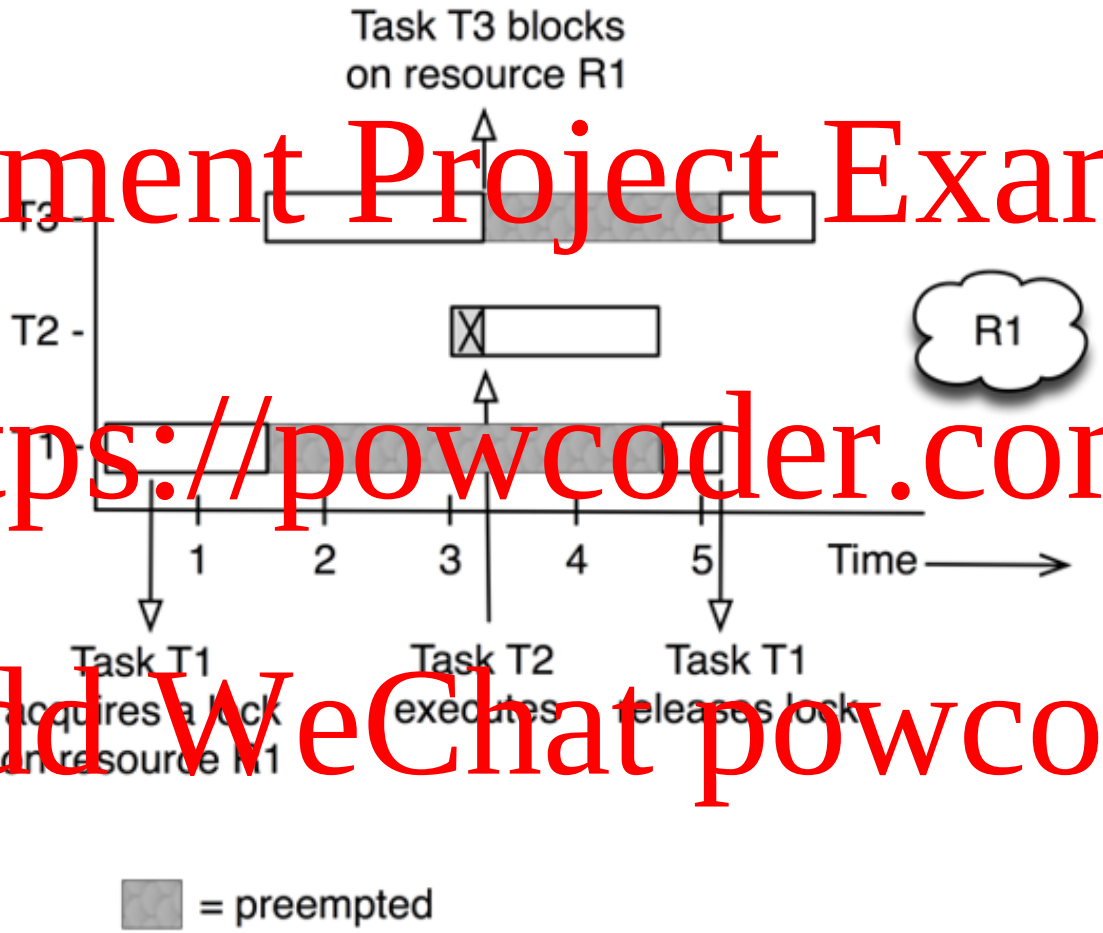


Figure: Priority Inversion

Common Scheduling Policies

- *First-in first-out (FIFO)* – process jobs in the order that they arrive in the ready queue
- *Fixed-priority scheduling* – assign each source of resource requests a particular priority and assign the resources in that priority order. Prioritization strategies include:
  - *Semantic importance* – Assign priority statically according to some domain characteristic of the task
  - *Deadline monotonic* – Assign higher priority to streams with shorter deadlines
  - *Rate monotonic* – Assign higher priority to streams with shorter periods
- *Dynamic priority scheduling*
  - *Round-robin* – Assign jobs to resources in a rotating fashion. *Cyclic executive* – Assign a fixed time unit per process, and cycle through them (requires preemption)
  - *Earliest-deadline-first* – Assign priorities based on the pending requests with the earliest deadline (optimal)
  - *Least-slack-first* – Assign the highest priority to the job with the least *slack time*, which is the difference between the execution time remaining and the time to the job's deadline (optimal)
- *Static scheduling* – the preemption points and sequence of assignment to the resource are determined offline (thus there is no runtime overhead for scheduling)

Table of Contents

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.
- Use specialized compute resources when available. GPUs are far faster than CPUs on many different computational tasks.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.
- Use specialized compute resources when available. GPUs are far faster than CPUs on many different computational tasks.
- Use tools like profilers. Consider embracing *causal profilers* like Coz.

Other Performance Topics

- Be data-driven. Religiously collect and analyze performance metrics and use these to drive your decisions.
- Understand and work with (not against) your CPU's branch predictor and memory prefetcher
  - Linear search is faster than binary search for small and medium sized arrays [Ano17]
  - Contiguous data structures (e.g. vectors) are often faster than non-contiguous ones (e.g. lists) regardless of what the theoretical big-O differences are.
- Use specialized compute resources when available. GPUs are far faster than CPUs on many different computational tasks.
- Use tools like profilers. Consider embracing *causal profilers* like Coz.
- Computer systems have a lot of noise which can make measuring the impact of individual performance improvements very difficult. A strong background in statistics is extremely helpful.

Optional Readings

- Don't Compare Averages
- How Shopify Reduced Storefront Response Times with a Rewrite
- Asynchronous computing @Facebook: Driving efficiency and developer productivity at Facebook scale
- Impact of Intel vs. ARM CPU Performance for Object Storage
- How we reduced latency and cost-to-serve by merging two systems
- Why are services slow sometimes?
- Introduction to Profiling and Optimizing SQL Queries for Software Engineers
- Understanding CPU Microarchitecture for Performance
- Reflections on software performance
- Memory Bandwidth Napkin Math
- Why Kafka Is so Fast

Table of Contents

- Read chapter 8 from SAIP
- Homework 2 is due Thursday, October 8 at 5:30 PM.
- Quiz 2 is availabile now. It is due Thursday, October 8 at 5:30 PM.

References

[Ano17] Anonymous.
Performance comparison: linear search vs binary search.
Dirty hands coding, 2017.

[BCK12] Len Bass, Paul Clements, and Rick Kazman.
Software Architecture in Practice.
Addison-Wesley Professional, 3rd edition, 2012.

[Bon12] Jonas Bonér.
Latency numbers every programmer should know, 2012.

[Chr10] Chris.
Microsoft affirms the importance of web performance.
Web Performance, 2010.

[Dix09] Philip Dixon.
Shopzilla's site redo - you get what you measure.
O'Reilly Velocity, 2009.

[Eat12] Kit Eaton.
How one second could cost amazon $1.6 billion in sales.
Fast Company, 2012.

[Knu74] Donald E. Knuth.
Structured programming with go to statements.
ACM Comput. Surv., 6(4):261–301, December 1974.

[Ros12] Matt Rosoff.
Google engineers holzle: speed is king.
Business Insider, 2012.