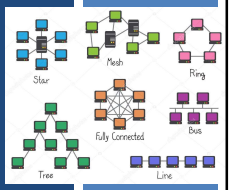
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Distributed Systems: Goals and Types

Wes J. Lloyd
School of Engineering and Technology
University of Washington - Tacoma



OBJECTIVES

- Feedback from 1/7
- Design goals of distributed systems
  - Resource sharing / availability
  - Distribution transparency
  - Openness
  - Scalability
- Types of distributed systems
  - HPC, cluster, grid, cloud
  - Distributed information systems
  - Pervasive systems
- Research directions
- Chapter 2: Distributed System Architectures

FEEDBACK – 1/7

- Key concepts from Jan 7th:
- **ACCESSIBILITY** in distributed systems
  - The idea of making resources accessible to users
  - Examples: GPU servers, FPGA servers
  - These are unique and expensive compute resources
  - The cloud makes these accessible via middleware (Amazon EC2 API)

FEEDBACK - 2

- Why is it called **DISTRIBUTION TRANSPARENCY**, if many of the aspects are obscured from the user?
  - In distributed systems, "hidden features" are considered to be transparent to users.
  - Transparent means "clear" or "lucid"
  - The details of the distribution are hidden in lower architectural layers (i.e. not exposed to users – they can't interact with the configuration) → the distribution is considered to be transparent

FEEDBACK - 3

- **FAILURE TRANSPARENCY**
  - Instances of HW or SW failure are hidden from users
- **POLICY vs. MECHANISM**
  - Policy – rules or business logic
  - Mechanism – technology used for system implementation
- **OPENNESS**
  - We will review again today…
- **SCALABILITY**
  - Covered today…

FEEDBACK - 4

- What is considered a good time for eventual consistency?
  - It depends on:
  - #1 – the number of replicas involved (how many)
  - #2 – the distribution of the replicas (how far apart)
  - #3 – the degree of connectedness of the replicas (networking)
  - Communication is generally limited by light speed
- See paper:
- Benchmarking Eventual Consistency - Lessons Learned from Long-Term Experimental Studies
  - Paper relates to NoSQL key-value store databases (e.g. AWS S3)

FEEDBACK - 5

- How many tutorials will we have?
  - Assignment 0 is very tutorial like
- Will midterm and final be open book?
  - Final – yes – book + notes
  - Midterm – TBD

DESIGN GOALS OF DISTRIBUTED SYSTEMS

- Support for sharing resources (accessibility)
- Distribution transparency
- Openness (avoiding vendor lock-in)
- Scalability

OPENNESS

- System with components that are easily used by, or integrated into other systems
- **Key aspects of openness:**
- Interoperability, portability, extensibility
- **Interfaces**: provide general syntax and semantics to interact with distributed components
- Services expose interfaces: functions, parameters, return values
- Semantics: describe what the services do
  - Often informally specified (via documentation)
- General interfaces enable alternate component implementations

OPENNESS - 2

- **Interoperability**: ability for components from separate systems to work together (different vendors?)
- Though implementation of a common interface
- How could we measure interoperability of components?
- **Portability**: degree that an application developed for distributed system A can be executed without modification on distributed system B
- How could we evaluate portability of a component?
- What percentage of portability is expected?

OPENNESS

- **Extensible**: easy to reconfigure, add, remove, replace components from different developers
- Example: replace the underlying file system of a distributed system
- To be open, we would like to ***separate policy from mechanism***
- Policy may change
- Mechanism is the technological implementation
- Avoid coupling policy and mechanism
- Enables flexibility
- Similar to separation of concerns, modular/OO design principle

SEPARATING POLICY FROM MECHANISM

- Example: **web browser caching**
- **Mechanism:** browser provides facility for storing documents
- **Policy:** Users decide which documents, for how long, …
- Goal: Enable users to set policies dynamically
- For example: browser may allow separate component plugin to specify policies
- **Tradeoff**: management complexity vs. policy flexibility
- Static policies are inflexible, but are easy to manage as features are barely revealed.
- AWS Lambda (Function-as-a-Service) abstracts configuration polices from the user resulting in management simplicity

OPENNESS EXAMPLE

- **Which of the following designs is more open?**
- Acme software corporation hosts a set of public weather web services (e.g. web service API)
- **DESIGN A:** API is implemented using MS .NET Remoting
- .NET Remoting is a mechanism for communicating between objects which are not in the same process. It is a generic system for different applications to communicate with one another. .NET objects are exposed to remote processes, thus allowing inter process communication. The applications can be located on the same computer, different computers on the same network, or on computers across separate networks.

OPENNESS EXAMPLE - 2

- **DESIGN B**: API is implemented using Java RMI
- The Java Remote Method Invocation (RMI) is a Java API that performs remote method invocation to allow Java objects to be distributed across different Java program instances on the same or different computers. RMI is the Java equivalent of C remote procedure calls, which includes support for transfer of serialized Java classes and distributed garbage-collection.
- **DESIGN C**: API is implemented as HTTP/RESTful web interface
- A RESTful API is an API that uses HTTP requests to GET, PUT, POST and DELETE data. RESTful APIs are referred to as a RESTful web services

TYPES OF SCALABILITY

- **Size scalability**: distributed system can grow easily ***without*** impacting performance
  - Supports adding new users, processes, resources
- **Geographical scalability**: users and resources may be dispersed, but communication delays are negligible
- **Administrative scalability:** Policies are scalable as the distributed system grows… (security, configuration management policies are agile enough to deal with growth) ***Goal: have administratively scalable systems !***
- Most systems only account for size scalability
- One solution is to operate multiple parallel independent nodes

SIZE SCALABILITY

- Centralized architectures have limitations
- At some point a single central coordinator/arbitrator node can't keep up
  - Centralized server: limited CPU, disk, network capacity
- Scaling requires surmounting bottlenecks

Lloyd W, Pallickara S, David O, Lyon J, Arabi M, Rojas K. Migration of multi-tier applications to infrastructure-as-a-service clouds: An investigation using kernel-based virtual machines. InGrid Computing (GRID), 2011 12th IEEE/ACM International Conference on 2011 Sep 21 (pp. 137-144). IEEE.

GEOGRAPHIC SCALABILITY

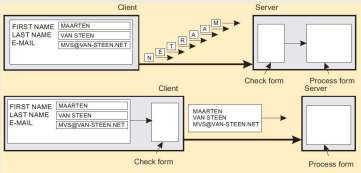
- Nodes dispersed by great distances
  - Communication is slower, less reliable
  - Bandwidth may be constrained
- How do you support synchronous communication?
  - Latencies may be higher
  - Synchronous communication may be too slow and timeout
  - WAN links can be unreliable

ADMINISTRATIVE SCALABILITY

- Conflicting policies regarding usage (payment), management, and security
- How do you manage security for multiple, discrete data centers?
- Grid computing: how can resources be shared across disparate systems at different domains, etc. ?

APPROACHES TO SCALING

- **Hide communication latencies**
  - Use asynchronous communication to do other work and hide latency
  - Remote server runs in parallel in the background – client not locked
  - Separate event handler captures return response from server
- Hide latency by moving key press validation to client:



APPROACHES TO SCALING - 2

- Partitioning data and computations across machines
- Just one copy
  - Where is the copy?
- Move computations to the client
  - Thin client → thick client
  - Edge, fog, cloud….
- Decentralized naming services (DNS)
- Decentralized information services (WWW)

APPROACHES TO SCALING - 3

- Replication and caching – make copies of data available at different machines
- Replicated file servers and databases
- Mirrored web sites
- Web caches (in browsers and proxies)
- File caches (at server and client)
- **LOAD BALANCER** (or proxy server)
  - Commonly used to distribute user requests to nodes of a distributed system

PROBLEMS WITH REPLICATION

- Having multiple copies leads to inconsistency (cached or replicated)
- Modifying one copy invalidates all of the others
- Keeping copies consistent requires global synchronization
- Global-synchronization prohibits large-scale up
  - Best to synchronize just a few copies or synchronization latency becomes too long, entire system slows down!
  - ***Consider how synchronization time increases with system size***
- Can these inconsistencies be tolerated?

1. Current temperature and wind speed from weather.com
2. Bank account balance – for a read only statement
3. Bank account balance – for a transfer/withdrawal transaction

DEVELOPING DISTRIBUTED SYSTEMS

- Developing a distributed system is a formidable task
- Many issues to consider:
- Reliable networks do not exist
- Networked communication is inherently insecure

FALSE ASSUMPTIONS ABOUT DISTRIBUTED SYSTEMS

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

TYPES OF DISTRIBUTED SYSTEMS:

HPC, CLUSTER, GRID, CLOUD

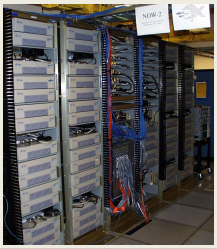
L2.25

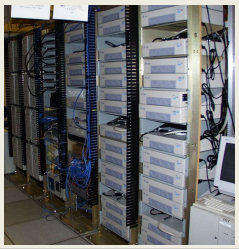
TECHNOLOGY INNOVATIONS LEADING TO CLOUD COMPUTING

- Super computers
  - Huge multiprocessor system which shares RAM
  - Technically "not distributed"
  - Hardware all in one location
- High performance distributed computing
  - Cluster computing
  - Grid computing
  - Cloud computing
  - Virtualization
  - Others

EARLY CLUSTER - 1996

- Inktomi search engine on Network of Workstations (NOW) @ UC Berkeley in 1996





CLUSTER COMPUTING

- Cluster computing (clustering)
  - Cluster is a group of independent IT resources interconnected as a single system
  - Off-the-shelf computers connected via a high-speed network
  - Servers configured with homogeneous hardware and software
    - Identical or similar RAM, CPU, HDDs
  - Design emphasizes redundancy as server components are easily interchanged to keep overall system running
    - Example: if a RAID card fails on a key server, the card can be swapped from another redundant server
  - Clusters provide "warm" replication of servers
    - Key servers are duplicated to provide HW failover to ensure high availability (HA)

COMPUTER CLUSTERS

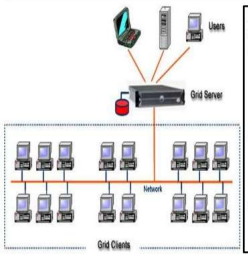
- Clusters: Commodity computers connected by Ethernet switches
  - More scalable than conventional servers
  - Much cheaper than conventional servers
  - Dependability through extensive redundancy
  - Few administrators for 1000s servers
- Careful selection of identical HW/SW
  - Interchangeable components
- Virtual Machine Monitors simplify operation

GRID COMPUTING

- On going research area since early 1990s
- Distributed heterogeneous computing resources organized into logical pools of loosely coupled resources
- For example: heterogeneous servers connected by the internet
- Resources are heterogeneous and geographically dispersed
- Grids use middleware software layer to support workload distribution and coordination functions
- Aspects: load balancing, failover control, autonomic configuration management
- Grids have influenced clouds contributing common features: networked access to machines, resource pooling, scalability, and resiliency

GRID COMPUTING - 2

How Grid computing works ?



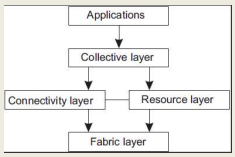
In general, a grid computing system requires:
- At least one computer, usually a server, which handles all the administrative duties for the System
- A network of computers running special grid computing network software.
- A collection of computer software called middleware

GRID COMPUTING - 3

- Grids are built by federating compute resources together from many organizations
- **Virtual organization**
  - Users from different organizations participate together in a virtual organization
  - Jobs belonging to a virtual organization can harness resources owned by the virtual organization
- Grids bring together heterogeneous hardware owned by many organizations

GRID COMPUTING LAYERS

- **Application layer**
  - Applications operating within a virtual organization sharing grid resources
- Middleware layers
  - **Collective layer**
    - Provides access to multiple resources
    - Services for discovery, allocation, scheduling, data replication, etc.
  - **Connectivity layer**
    - Communication protocols to support transactions across grid
    - Data transfer, access to resources, security (authentication) protocols
  - **Resource layer**
    - Manages access to a single resource via fabric layer
    - Configuration of a specific resource
    - Security (access control)
- **Fabric layer**

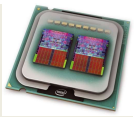


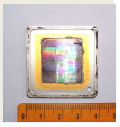
CLOUD COMPUTING NIST GENERAL DEFINITION

"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (networks, servers, storage, applications and services) that can be rapidly provisioned and reused with minimal management effort or service provider interaction"…

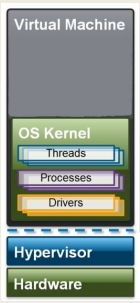
MICROPROCESSORS ADVANCEMENTS

- Smaller die sizes (microns)
  - Lower voltages
  - Improved heat dissipation
  - Energy conservation
  - More transistors, but with similar clock rates
- Leads to multicore CPUs
  - Means to harness new transistor density
  - Improve overall computational throughput
- How do we utilize many-core processors?

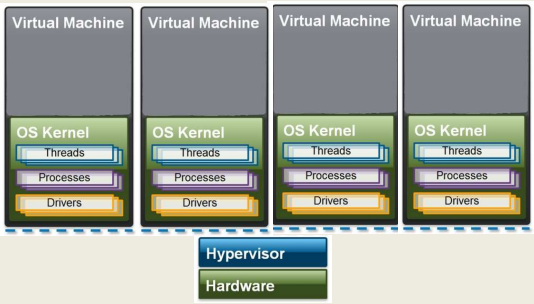




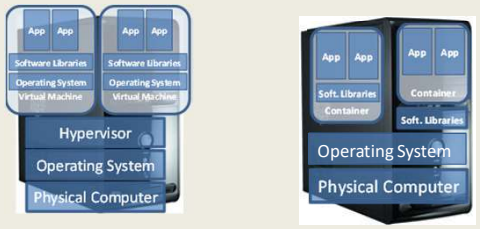
VIRTUALIZATION



VIRTUALIZATION



CONTAINERIZATION

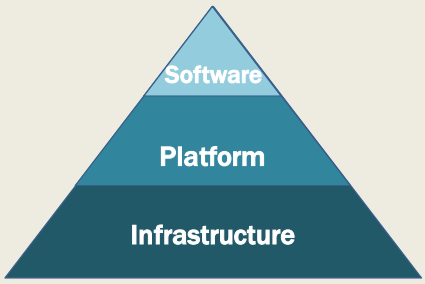


Virtualization **Containerization**

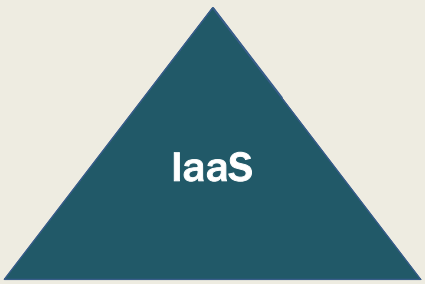
HOW WAREHOUSE SCALE COMPUTING BECAME THE CLOUD

- Clusters grew from 1,000 servers to 100,000+ based on customer demand for SaaS apps
- Economies of scale pushed down costs by 3X to 8X
  - Purchase, house, operate 100K vs. 1K computers
  - Traditional datacenters utilization is ~ 10% - 20%
- Earn $ offering pay-as-you-go computing at prices lower than customer's costs;
  - Scalable → as many computers as customer needs

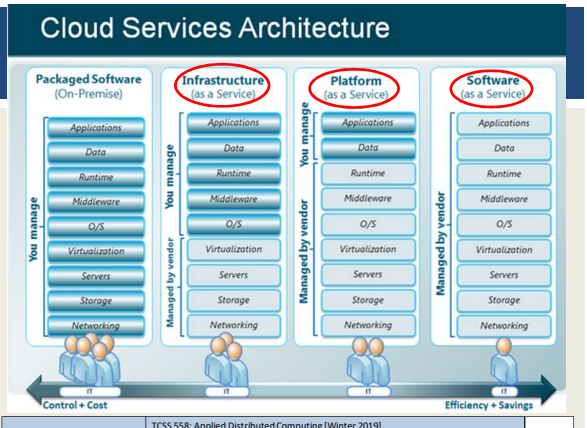
CLOUD COMPUTING STACK



CLOUD COMPUTING STACK



Cloud Services Architecture



PUBLIC CLOUD EXAMPLE: NETFLIX

- Amazon Elastic Compute Cloud (EC2)
  - Continuously run 20,000 to 90,000 VM instances
  - Across 3 regions
  - Host 100s of microservices
  - Process over 100,000 requests/second
  - Host over 1 billion hours of monthly content

NETFLIX

PUBLIC CLOUD COMPUTING

- Offers computing, storage, communication at ¢ per hour
- No premium to scale:

```
1000 computers   @   1 hour
=     1 computer     @ 1000 hours
```

- Illusion of infinite scalability to cloud user
- As many computers as you can afford
- Leading examples:
  Amazon Web Services, Google App Engine, Microsoft Azure
- Amazon runs its own e-commerce on AWS!
- Billing models are becoming increasingly granular
  - By the minute, second, tenth of a second
  - Obfuscated pricing-Lambda $0.0000002 per request
    $0.000000208 to rent 128MB / 100-ms

PUBLIC CLOUD COMPUTING

m4.large ec2 virtual machine:
2 vCPU cores, 8 GB RAM, Intel Xeon E5-2666 v3
10¢ an hour, 24 hrs/day,
30 days/month → $72.00/month
on-demand EC2 instance

AWS Lambda Function-as-a-Service (FaaS):
2 vCPU cores, 3GB RAM, Intel Xeon E5-2666 v3
as 2,592,000 x 1-sec service calls
24 hrs/day, 30 days/month:

$130.14 (8GB = $347.04)

PAAS SERVICES IMPLEMENTATION

- PaaS services often built atop of IaaS
  - Amazon RDS, Heroku, Amazon Elasticache
- Scalability
  - VM resources can support fluctuations in demand
- Dependability.
  - PaaS services built on highly available IaaS resources

TYPES OF DISTRIBUTED SYSTEMS:

DISTRIBUTED INFORMATION SYSTEMS

DISTRIBUTED INFORMATION SYSTEMS

- Enterprise-wide integrated applications
  - Organizations confronted with too many applications
  - Interoperability among applications was difficult
  - Lead to many middleware-based solutions
- Key concepts
  - Component based architectures - database components, processing components
  - **Distributed transaction** – Client wraps requests together, sends as single aggregated request
  - Atomic: ***all*** or ***none*** of the individual requests should be executed
- Different systems define different ***action*** primitives
  - Components of the atomic transaction
  - Examples: send, receive, forward, READ, WRITE, etc.

DISTRIBUTED INFORMATION SYSTEMS - 2

- Transaction primitives

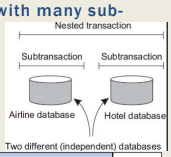
Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

- Transactions are all-or-nothing
  - All operations are executed
  - None are executed

TRANSACTIONS: ACID PROPERTIES

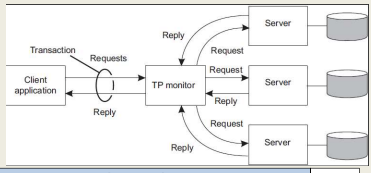
- **A**tomic: The transaction occurs indivisibly
- **C**onsistent: The transaction does not violate system invariants
  - Replicas remain constant until all updated
- **I**solated: Transactions do not interfere with each other
- **D**urable: Once a transaction commits, change are permanent

- Nested transaction: transaction constructed with many sub-transactions
- Follows a logical division of work
- Must support "rollback" of sub-transactions



TRANSACTION PROCESSING MONITOR

- Allow an application to access multiple DBs via a transactional programming model
- **TP monitor**: coordinates commitment of sub-transactions using a distributed commit protocol (Ch. 8)
- Save application complexity from having to coordinate



ENTERPRISE APPLICATION INTEGRATION

- Support application components direct communication with each other, not via databases
- **Communication mechanisms**:
- Remote procedure call (RPC)
  - Local procedure call packaged as a message and sent to server
  - Supports distribution of function call processing
- **Remote method invocations** (RMI)
  - Operates on objects instead of functions
- RPC and RMI – lead to tight coupling
- Client and server endpoints must be up and running
- Interfaces not so interoperable
- Leads to **Message-oriented middleware** (MOM)

MESSAGE-ORIENTED MIDDLEWARE

- Publish and subscribe systems:
  - Rabbit MQ, Apache Kafka, AWS SQS
- Reduces tight coupling of RPC/RMI
- Applications indicate interest for specific type(s) of message by sending requests to logical contact points
- Communication middleware delivers messages to subscribing applications

CHALLENGES WITH VARIOUS APPLICATION INTEGRATION METHODS

- File transfer
  - Shared data files (e.g. XML)
  - Leads to file management challenges
- Shared database
  - Centralized DB, transactions to coordinate changes among users
  - Common data schema required – can be challenging to derive
  - For many reads and updates, shared DB becomes bottleneck
- Remote procedure call – app A executes on and against app B data. App A lacks direct access to app B data.
- Messaging middleware - ensures nodes temporarily offline later can receive messages

TYPES OF DISTRIBUTED SYSTEMS:

PERVASIVE SYSTEMS

L2.55

PERVASIVE SYSTEMS

- Existing everywhere, widely adopted…
- Combine current network technologies, wireless computing, voice recognition, internet capabilities and AI to create an environment where connectivity of devices is embedded, unobtrusive, and always available
- Many sensors infer various aspects of a user's behavior
  - Myriad of actuators to collect information, provide feedback
- **TYPES OF PERVASIVE SYSTEMS:**
- Ubiquitous computing systems
- Mobile systems
- Sensor networks

PERVASIVE SYSTEM TYPE: UBIQUITOUS COMPUTING SYSTEMS

- Pervasive and continuously present
- Goal: embed processors everywhere (day-to-day objects) enabling them to communicate information
- Requirements for a ubiquitous computing system:
  - **Distribution** – devices are networked, distributed, and accessible transparently
  - **Interaction** – unobtrusive (low-key) between users and devices
  - **Context awareness** – optimizes interaction
  - **Autonomy** – devices operate autonomously, self-managed
  - **Intelligence** – system can handle wide range of dynamic actions and interactions

UBIQUITOUS COMPUTING SYSTEM EXAMPLE

- **Domestic ubiquitous computing environment example:**
- Interconnect lighting and environmental controls with personal biometric monitors woven into clothing so that illumination and heating conditions in a room might be modulated, continuously and imperceptibly
- IoT technology helps enable ubiquitous computing

PERVASIVE SYSTEM TYPE: MOBILE SYSTEMS

- Emphasis on mobile devices, e.g. smartphones, tablet computers
- New devices: remote controls, pagers, active badges, car equipment, various GPS-enabled devices,
- Devices move, where is the device?
- Changing location: leverage mobile adhoc network (MANET)
- MANET is an ad hoc network that can change locations and configure itself on the fly. MANETS are mobile, they use wireless connections to connect to various networks.
- VANET (Vehicular Ad Hoc Network), is a type of MANET that allows vehicles to communicate with roadside equipment.

PERVASIVE SYSTEM TYPE: SENSOR NETWORKS

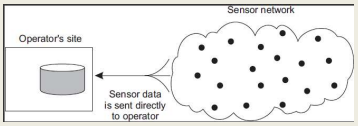
- Tens, to hundreds, to thousands of small nodes
- Simple: small memory/compute/communication capacity
- Wireless, battery powered (or battery-less)
- Limited: restricted communication, constrained power
- Equipped with sensing devices
- Some can act as actuators (control systems)
  - Example: enable sprinklers upon fire detection
- Sensor nodes organized in neighborhoods
- Scope of communication:
  - Node – neighborhood – system-wide

PERVASIVE SYSTEM TYPE: SENSOR NETWORKS - 2

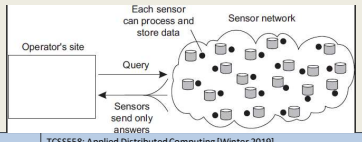
- Collaborate to process sensor data in app-specific manner
- Provide mix of data collection and processing
- **Nodes may implement a distributed database**
- Database organization: centralized to decentralized
- In network processing: forward query to all sensor nodes along a tree to aggregate results and propagate to root
- Is aggregation simply data collection?
- Are all nodes homogeneous?
- Are all network links homogeneous?
- How do we setup a tree when nodes have heterogeneous power and network connection quality?

CENTRALIZED VS. DECENTRALIZED DATA STORAGE

- Centralized:



- Decentralized:



WHO AGGREGATES AND STORES DATA?

- Consider the tradeoff space for:
  - sensor network data storage and processing

Centralized ⟷ **Decentralized**

- Single point-of-failure
- No node coordination
- No node processing or storage
- "Dumb" nodes
- Less expensive node
- More network traffic
- Nodes require high compute power
- "Smart" nodes
- Expensive nodes
- Less network traffic

SENSOR NETWORKS - 3

- What are some unique requirements for sensor networks middleware?
  - Sensor networks may consist of different types of nodes with different functions
  - Nodes may often be in suspended state to save power
    - Duty cycles (1 to 30%), strict energy budgets
  - Synchronize communication with duty cycles
  - How do we manage membership when devices are offline?

RESEARCH DIRECTIONS

THIS WINTER

- **Research group meetings**
  - Cloud/Distributed Sys - Tuesdays 12:00-1:30pm, MDS 312
  - Bioinformatics – Wednesday 11:30-1:00pm, TLB 307C
- **Goals**:
  - Assemble ongoing agile research teams which maximize opportunities for student collaboration and sharing to lower the bar for student engagement in research
  - Build on past successes through iterative student contributions
  - Maximize student learning and research outcomes
  - Provide students a practicum in cloud computing research to increase competitiveness in industry and graduate school

Serverless Computing

Image from: https://mobisoftinfotech.com/resources/blog/serverless-computing-deploy-applications-without-fiddling-with-servers/

67

SERVERLESS COMPUTING

- Pay only for CPU/memory utilization
- High Availability
- Fault Tolerance
- Infrastructure Elasticity
- No Setup
- Function-as-a-Service (FAAS)

SERVERLESS COMPUTING

Why Serverless Computing?

Many features of distributed systems, that are challenging to deliver, are provided automatically

…they are built into the platform

SERVERLESS COMPUTING

- Refers to the avoidance of managing servers
- Serverless can pertain to a variety of cloud services
- Evolving technology
  - Function-as-a-Service (FaaS)
  - Database-as-a-Service (DBaaS)
    - Amazon Aurora Serverless DB – general availability Aug 9
  - Container-as-a-Service (CaaS)
    - Google Kubernetes Engine serverless add-on
  - Others…

FAAS PLATFORMS

- AWS Lambda
- Azure Functions
- IBM Cloud Functions
- Google Cloud Functions

Commercial

Open Source

- Apache OpenWhisk
- Fn (Oracle)

SERVERLESS COMPUTING

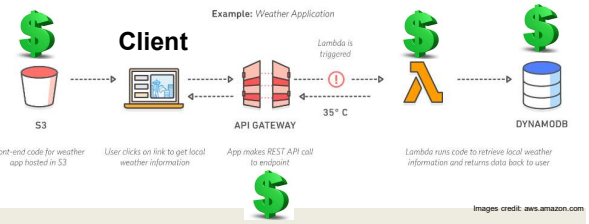
Research Challenges

Image from: https://mobisoftinfotech.com/resources/blog/serverless-computing-deploy-applications-without-fiddling-with-servers/

72

VENDOR ARCHITECTURAL LOCK-IN

- Cloud native (FaaS) software architecture requires external services/components



- Increased dependencies → increased hosting costs

PRICING OBFUSCATION

- **VM pricing:** hourly rental pricing, billed to nearest second is intuitive…
- **FaaS pricing:** non-intuitive pricing policies
- **FREE TIER:**
  first 1,000,000 function calls/month → FREE
  first 400,000 GB-sec/month → FREE
- Afterwards: *obfuscated pricing (AWS Lambda):*
  $0.0000002 per request
  $0.000000208 to rent 128MB / 100-ms
  $0.00001667 GB /second

WEBSERVICE HOSTING EXAMPLE

- **ON AWS Lambda**
- Each service call: 100% of 1 CPU-core
  100% of 4GB of memory
- Workload: 2 continuous client threads
- Duration: 1 month (30 days)

- **ON AWS EC2:**
- Amazon EC2 c4.large 2-vCPU VM
- Hosting cost: $72/month
  c4.large: 10¢/hour, 24 hrs/day x 30 days

- **How much would hosting this workload cost on AWS Lambda?**

PRICING OBFUSCATION

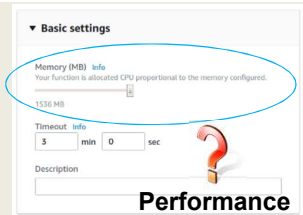
Worst-case scenario = ~2.32x !

AWS EC2:	$72.00
AWS Lambda:	$167.01
Break Even:	4,319,136 GB-sec
Two threads @2GB-ea:	~12.5 days

- **BREAK-EVEN POINT: ~4,319,136 GB-sec-month**
  ~12.5 days 2 concurrent clients @ 2GB

MEMORY RESERVATION QUESTION…

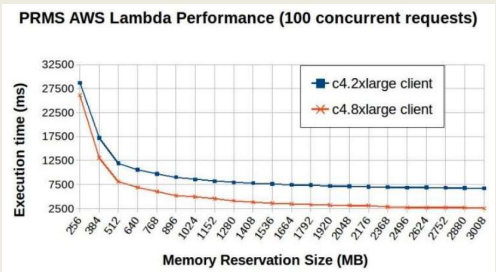
- Lambda memory reserved for functions
- UI provides "slider bar" to set function's memory allocation
- Resource capacity (CPU, disk, network) coupled to slider bar:
  "*every **doubling** of memory, **doubles** CPU…*"



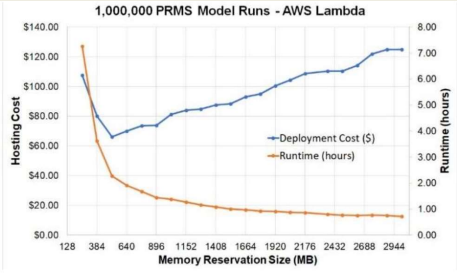
- **But how much memory do model services require?**

LAMBDA: PERFORMANCE VS MEMORY

- Order of magnitude performance gain ~ 10x



HOW MUCH FOR 1,000,000 CALLS?



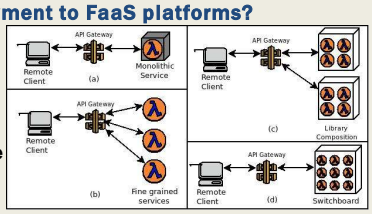
CLOUD NATIVE APPLICATIONS: EVOLVING BEST PRACTICES

- Coupling between classes/modules
  - Degree dependence between software modules
  - Measure of how closely connected two modules are
- Cohesion between classes/modules
  - Strength of relationships between methods and data
  - How unified is the purpose or concepts of groupings
  - Functional cohesion
- Object-Oriented Software Best Practice:
  Minimize Coupling, Maximize Cohesion
- Shown to correlate with software quality:
  maintainability, reusability, extensibility, understandability

SERVICE COMPOSITION

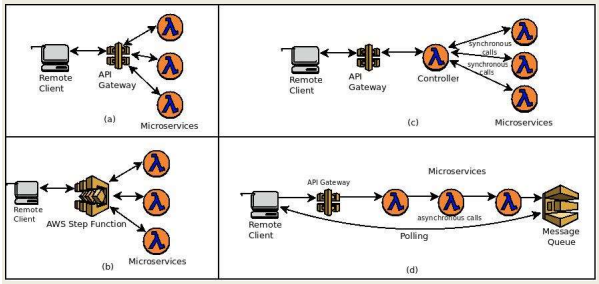
- **How should application code be composed for deployment to FaaS platforms?**

Performance



- Best practice: decompose into many microservices
- Platform limits: code + libraries ~250MB
- **How does FaaS function composition impact performance and cost of native cloud applications?**

APPLICATION FLOW CONTROL



INFRASTRUCTURE FREEZE/THAW CYCLE

- Unused infrastructure is deprecated
  - *But after how long?*
- Infrastructure: VMs, "containers"
- **Provider-COLD / VM-COLD**
  - "Container" images - built/transferred to VMs
- **Container-COLD**
  - Image cached on VM
- **Container-WARM**
  - "Container" running on VM

Performance

Image from: Denver7 – The Denver Channel News

SUMMARY OF FAAS CHALLENGES

- Vendor architectural lock-in – how to migrate?
- Pricing obfuscation – is it cost effective?
- Memory reservation – how much to reserve?
- Service composition – how to compose software?
- App flow control – implications of implementation?
- Infrastructure freeze/thaw cycle – how to avoid?
- Platform constraints – memory, runtime, codesize

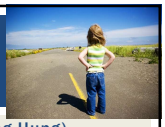
SERVERLESS COMPUTING

- **Microservices**:
- Service composition – *Baojia Zhang*
  - *Performance and cost implications of microservice disaggregation vs. composition*
- FaaS Application Migration – *Baojia Zhang*
- FaaS Platform Simulation and Modeling – *Lan Ly*
- Freeze/Thaw Lifecycle Mitigation – *Minh Vu*
- Cloud vs Edge vs Device – *Harrison Ross*
- Leveraging Serverless Computing for Computer Vision Neural Networks – *Vlad Kaganyuk (t-mobile)*
- FaaS Inspector Toolkit - *Shruti Ramesh (Microsoft)* https://github.com/wlloyduw/faas_inspector

CONTAINERIZATION

- Application system containers - Docker
- Container orchestration framework(s) – Kubernetes, Docker Swarm, Apache Mesos/Marathon, AWS Elastic Container Service
- T-Mobile Container Platform Study– *Garrett Lahmann*
- Analyzing the gap between resource reservation and utilization on container platforms
- Workflow Containerization: Resource profiling of Docker containers - *Huazeng Deng*
- https://github.com/wlloyduw/ContainerProfiler
- Project extensions: integrate with Prometheus, Grafana

INFRASTRUCTURE-AS-A-SERVICE CLOUD RESEARCH



- Bioinformatics (w/ Kayee Yeung-Rhee, Ling-Hong Hung)–
  - Workflow scheduling - *Zelun "Jim" Jiang*
  - Container checkpointing - *Pai Zhang*
- eScience Institute (UW Seattle)
  - Rosetta (protein folding) – *Srihari Vignesh*
- Cloud vs. Edge for mobile computing workloads – *Harrison Ross*
- Intelligent deployment of bioinformatics workflows on the cloud to improve performance and cost
  - Performance benchmarking *Radhika Sridhar, Saranya Ravishankar*
  - Resource utilization profiling *Radhika Sridhar*
  - Performance Modeling, Machine Learning
- Infrastructure management improvements
  - **Public cloud resource contention and avoidance** – *Edward Han, Jugal Gandhi*

IAAS CLOUD - 2

- Infrastructure-as-a-Service Cloud Application Deployment
  - Performance modeling
    - Models to predict performance of alternate deployment schemes
  - Cost modeling
    - Models to predict costs of alternative deployment schemes
    - ➡ What is the best infrastructure for my workload?
    - ➡ What is the cost of deployment?
    - ➡ Should I migrate to containers, serverless computing?
- Reverse engineering of IaaS, PaaS, SaaS
  - ➡ What service level is best for my workload?

REVERSE ENGINEERING

- Clouds abstract infrastructure implementation from end users
- ***Design goal of distributed systems – transparency***
- Users access abstract infrastructure via software services
  - **As-a-service**: IaaS, PaaS, SaaS, FaaS, DBaaS, CaaS, cache services, storage, NoSQL-databases
- How do we best leverage abstract infrastructure?
- What performance and cost implications result from ignoring abstraction?
- What "value" does the service really provide? Is it worth it?
- What can we infer about abstract infrastructure that can help the users of cloud services? (*cloud consumers*)

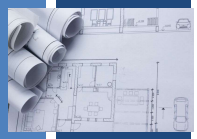
CLOUD FEDERATION / ENERGY

- Cloud federation and resource abstraction
  - How can we dynamically harness resources from diverse clouds to enable cost savings and high availability improvements?
  - Containers are a key enabling technology for platform independence
    - Bioinformatics applications
- Support green computing goals:
  - Opportunistic workload consolidation and migration to the most sustainable, economical, and energy efficient resources, ***T-Mobile***

VIRTUALIZATION / UNIKERNELS

- Lightweight alternative to containers and VMs
  - Custom Cloud Operating System
  - No process, multiple threads, run one program
  - Launch separately atop of hypervisor (XEN)
  - Reduce overhead, duplication of heavy weight OS
  - Performance comparison to containers, virtual machines
  - Web application (services) and native Java application comparison - ***Devin Durham***
  - Java Spring boot microservices on unikernels vs. containers
  - Much lower launch latency
  - Application performance is variable

CH. 2: DISTRIBUTED SYSTEMS ARCHITECTURES

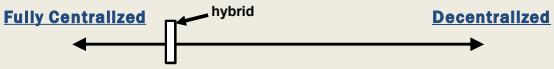


DISTRIBUTED SYSTEM ARCHITECTURES

- Logical organization of a distributed system into software components
- Logical: How system is perceived, modeled
  - *The OO/component abstractions*
- Physical – how it really exists
- Middleware
  - Helps separate application from platforms
  - Helps organize distributed components
  - How are the pieces assembled?
  - How do they communicate?
  - How are systems extended? replicated?
  - Provides "realization" of the architecture

CENTRALIZED VS. DECENTRALIZED DISTRIBUTED SYSTEM ARCHITECTURE

- Tradeoff space: degree of distribution of the system



• Single point-of-failure	• Multiple failure points
• No nodes: vertical scaling	• Nodes: horizontal scaling
• Always consistent	• Eventually consistent
• Less available (fewer 9s)	• More available (more 9s)
• Immediate updates	• Rolling updates
• No data partitions	• Data partitioned or replicated

ARCHITECTURAL BUILDING BLOCKS

- **Component**: modular unit with well-defined, required, and provided **Interfaces** that is replaceable within its environment
- Components can be replaced while system is running
- Interfaces must remain the same
- Preserving interfaces enables interoperability
- **Connector**: enables flow of control and data between components
- Distributed system architectures are conceived using components and connectors

ARCHITECTURAL STYLES

- Layered
- Object-based
  - Service oriented architecture (SOA)
- Resource-centered architectures
  - Representational state transfer (REST)
- Event-based
  - Publish and subscribe (Rich Site Summary RSS feeds)

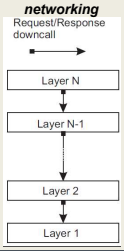
LAYERED ARCHITECTURES

- Components organized in layers
- Component at layer L_j downcalls to lower-level components at layer L_i (where i < j)
- Calls go down
- Exceptional cases may produce upcalls

LAYERED ARCHITECTURES - 2

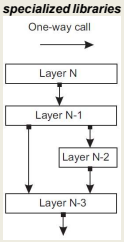
Pure-layered Organization

networking



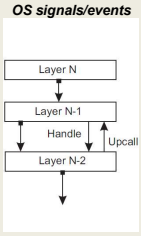
Mixed-layered organization

specialized libraries



Layered w/ upcalls organization

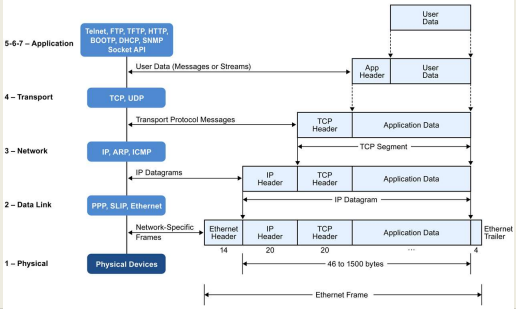
OS signals/events



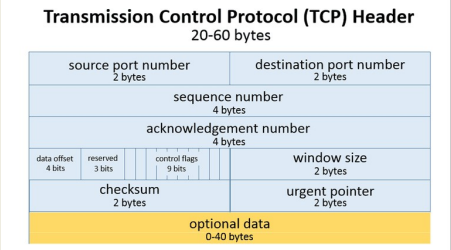
COMMUNICATION-PROTOCOL STACKS

- Example: pure-layered organization
- Each layer offers an interface specifying functions of the layer
- Communication protocol: rules used for nodes to communicate
- Layer provides a **service**
- **Interface** makes service available
- **Protocol** implements communication for a layer
- New services can be built atop of existing layers to reuse low level implementation
- Abstractions make it easier reuse existing layers which already implement communication basics

HOW A NETWORK PACKET IS BUILT

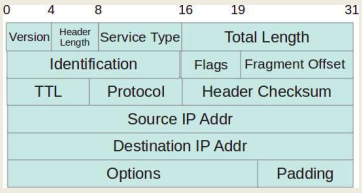


TCP HEADER



IP HEADER

- Source / Destination IP Addr
- IPv4: 32bits / 4 bytes
- IPv6: 128bits / 16 bytes



TRANSMISSION CONTROL PROTOCOL (TCP)

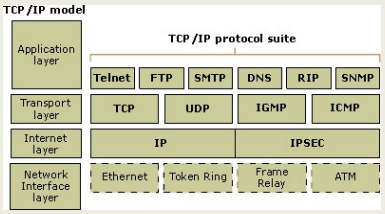
- TCP provides easy to use API
- API supports: setup, tear down of connection(s)
- API supports: sending and receiving of messages
- TCP preserves ordering of transferred data
- TCP detects and corrects lost data

- But TCP is "protocol" agnostic
  - E.g. language agnostic

- What are we going to say?

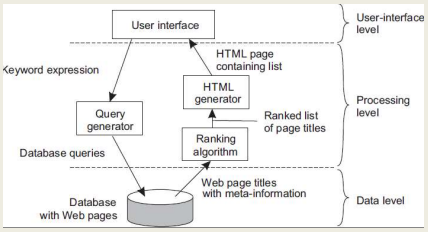
COMMON APPLICATION LAYER PROTOCOLS

- Telnet, FTP, TFTP, HTTP, DHCP, DNS, NTP, POP, RTP, SMTP, Telnet, RPC, LDAP



APPLICATION LAYERING

- Distributed application example: Internet search engine



APPLICATION LAYERING

- Three logical layers of distributed applications
  - The data level
  - Application interface level
  - The processing level

APPLICATION LAYERING

- Three logical layers of distributed applications
  - The data level (M)
  - Application interface level (V)
  - The processing level (C)

- Model view controller architecture – distributed systems
  - Model – database - handles data persistence
  - View – user interface - also includes APIs
  - Controller – middleware / business logic

OBJECT-BASED ARCHITECTURES

- Enables loose and flexible component organization
- Objects == components
- Enable distributed node interaction via function calls over the network
- Began with C - Remote Procedure Calls (RPC)
  - Straightforward: package up function inputs, send over network, transfer results back
  - Language independent
  - In contrast to web services, RPC calls originally were more intimate in nature
  - Procedures more "coupled", not as independent
  - The goal was not to decouple and widgetize everything

OBJECT-BASED ARCHITECTURES - 2

- Distributed objects Java- Remote Method Invocation (RMI)
  - Adds object orientation concepts to remote function calls
  - Clients bind to proxy objects
  - Proxy provide an object interface which transfers method invocation over the network to the remote host
- How do we replicate objects?
  - Object marshalling – serialize data, stream it over network
  - Unmarshalling- create an object from the stream
  - Unmarshall local object copies on the remote host
  - JSON, XML are some possible data formats

DISTRIBUTED OBJECTS

- A counterintuitive features is that state is not distributed
- Each "remote object" maintains its own state
- Remote objects may not be replicated
- Objects may be "mobile" and move around from node to node
  - Common for data objects
- For distributed (remote) objects consider
  - Pass by value
  - Pass by reference

SERVICE ORIENTED ARCHITECTURE

- Services provide always-on encapsulated functions over the internet/web
- Leverage redundant cloud computing infrastructure
- Services may:
  - Aggregate multiple languages, libraries, operating systems
  - Include (wrap) legacy code
- Many software components may be involved in the implementation
  - Application server(s), relational database(s), key-value stores, in memory-cache, queue/messaging services

SERVICE ORIENTED ARCHITECTURE - 2

- Are more easily developed independent and shared vs. systems with distributed object architectures
- Less coupling
- An error while invoking a distributed object may crash the system
- An error calling a service (e.g. mismatching the interface) generally does not result in a system crash

QUESTIONS

EXTRA SLIDES

FEEDBACK – 9/28

- What is the difference between extensibility and scalability?
  - Extensibility – ability for a system implementation to be extended with additional functionality
  - Scalability – ability for a distributed system to scale (up or down) in response to client demand
- What is the loss of availability in a distributed system?
  - Availability refers to "uptime"
  - How many 9s
  - (1 – (down time/ total time)) * 100%
- Transparency: term is confusing
  - Generally means "exposing everything", obfuscation is better
  - Distribution transparency means the implementation of the distribution cannot be seen

FEEDBACK - 2

- What do we mean by replication transparency?
  - Resources are automatically replicated (by the middleware/framework)
  - That fact that the distributed system has replica nodes is unbeknownst to the users
- How does replication improve system performance?
  - By replicating nodes, system load is "distributed" across replicas
  - Distributed reads – many concurrent users can read
  - Distributed writes – when replicating data, requires synchronization of copies

RESEARCH DIRECTIONS

- Serverless Computing: FaaS, CaaS, DBaaS
- Containerization, Container Platforms
- Infrastructure-as-a-Service (IaaS) Cloud
- Resource profiling, Measurement, Cloud System Data Analytics
- Application performance and cost modeling
- Autonomic infrastructure management to optimize cost and performance
- Cloud Federation, Workload Consolidation, Green Computing
- Virtualization / Unikernel operating systems
- **Domains:**
- Bioinformatics (genomic sequencing)
- Environmental modeling (USDA, USGS modeling applications)