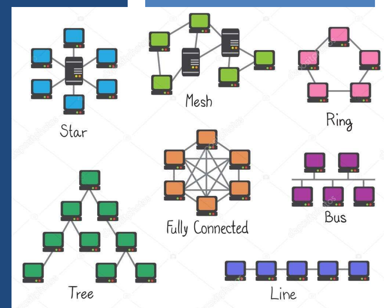
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Distributed Systems: Types and Architectures

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



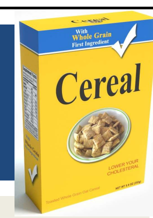
OBJECTIVES

- Homework 0 Posted
- Feedback from 1/14
- Types of distributed systems
  - HPC, cluster, grid, cloud
  - Distributed information systems
  - Pervasive systems
- Chapter 2: Distributed System Architectures
  - Architectural styles: Layered, Object-based, Resource-centered architectures, Event-based
- Research directions
  - Introduction to Serverless Computing
  - Containerization
  - Infrastructure-as-a-Service

FEEDBACK – 1/14

- **What is the difference between RPC and RMI?**
  - **RPC** is remote procedure call, originally for modular (non-object oriented) languages.
  - Idea is to remotely invoke C functions on remote servers
  - Parameters to make a local procedure call are “packaged up” and sent over the network
  - **RMI** is remote method invocation in Java
  - Servers host object instances
  - Java applications can invoke methods of “remote” objects over the network
  - **BOTH** provide abstraction as to where the actually code runs
  - **BOTH** require intimate knowledge of the precise function and object interfaces of remote resources

FEEDBACK - 2



- **CORBA** – Common object request broker architecture
- Provides a cross-language equivalent to RPC/RMI
- Languages: Ada, C, C++, C++11, COBOL, Java, Lisp, PL/I, Object Pascal, Python, Ruby and Smalltalk
- **RPC/RMI/CORBA**
  - Generally considered legacy technologies
- **Serialization:** RPC/RMI/CORBA technologies transfer data between nodes over the network.
- Network connections are byte streams
- Serialization is the “flattening” of classes and data structures (arrays) for transport over a byte stream

DESIGN GOALS OF DISTRIBUTED SYSTEMS

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

TYPES OF DISTRIBUTED SYSTEMS

- HPC, Cluster, Grid, Cloud
- Distributed information systems
- Pervasive Systems

TYPES OF DISTRIBUTED SYSTEMS:

DISTRIBUTED INFORMATION SYSTEMS

L4.7

FEEDBACK FROM 1/14

- Concept review:
- PaaS systems often implemented atop of IaaS
- Distributed systems use transactions
- Distributed transactions should follow ACID principles
  - A – Atomic: transaction occurs indivisibly
  - C – Consistent: replicas are consistent until all updated
  - I – Isolated: transactions don't interfere with each other
  - D – Durable: change is permanent committed
- Nested transaction - building transactions as set of sub-transactions

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.8

REVIEW - 2

- TP Monitor – Transaction Processing Monitoring
  - Facilitates implementation of the transaction across the nodes of the distributed system
  - TP monitor may be centralized component
- Methods for node-to-node communication
  - RPC/RMI – tight coupling to program code
  - REST services
  - MOM – message oriented middleware
    - Publish/subscribe queues
    - Supports message delivery for asynchronous (off-line) communication

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

COMMUNICATION

- Synchronous node communication
  - Channel remains open for duration of transaction
- Asynchronous node communication
  - Message is sent to initiate work, channel closed
  - Result is obtained via polling, or message exchange from a message queue or storage facility (database or key-value store)

January 16, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L4.11

TYPES OF DISTRIBUTED SYSTEMS:

PERVASIVE SYSTEMS

L4.12

PERVASIVE SYSTEMS

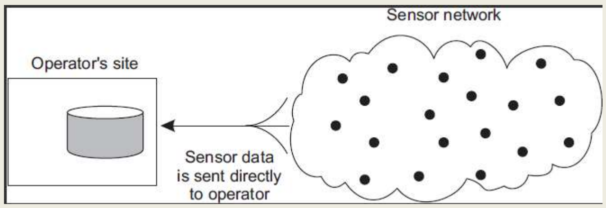
- **Ubiquitous computing systems**
  - Emphasis on integrating many heterogeneous devices to build cohesive collaborative systems
  - Example: IoT systems that provide new levels of intelligence by integrating multiple sources of data to control/manage environment (e.g. heating, cooling)
- **Mobile systems**
  - Emphasis on smartphones, tables, vehicles
  - Devices are physically mobile
  - Requires ad hoc networks to inter-node communication

PERVASIVE SYSTEMS - 2

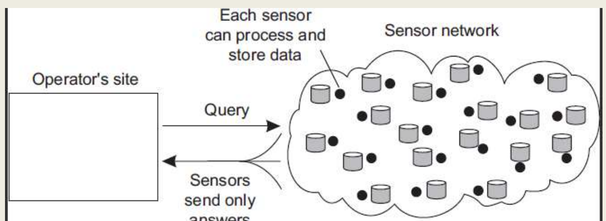
- **Sensor networks**
  - 10 – 100 – 1000s of small nodes with varying memory/compute/communication capacity
  - Different nodes collect different types of data
  - Issues regarding how to transport data to the cloud
  - Is all of the data needed?
  - Can aggregate data on the device an send preprocessed results upstream
  - Sensor network rely on unreliable adhoc networks
    - Node battery failure may cause network reconfiguration

CENTRALIZED VS. DECENTRALIZED DATA STORAGE

- Centralized:



- Decentralized:



January 16, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L4.15

WHO AGGREGATES AND STORES DATA?

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

Centralized Decentralized

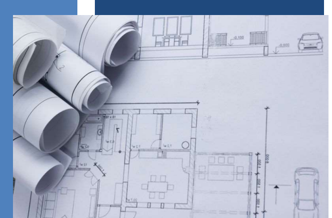
- Single point-of-failure
- No node coordination
- No node processing or storage
- "Dumb" nodes
- Less expensive node
- More network traffic

- Nodes require high compute power
- "Smart" nodes
- Expensive nodes
- Less network traffic

January 16, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L4.16

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?



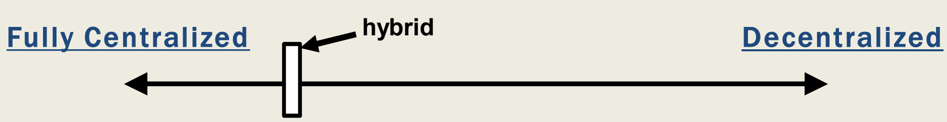
CH. 2: DISTRIBUTED SYSTEMS ARCHITECTURES

DISTRIBUTED SYSTEM ARCHITECTURES

- Provides logical organization of a distributed system into software **components**
- **Logical**: How system is perceived, modeled
  - *Object-oriented and 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

DISTRIBUTED SYSTEM ARCHITECTURES: CENTRALIZED VS. DECENTRALIZED

- **Tradeoff space**: degree of distribution of the system



- Single point-of-failure
- No nodes: vertical scaling
- Always consistent
- Less available (fewer 9s)
- Immediate updates
- No data partitions

- Multiple failure points
- Nodes: horizontal scaling
- Eventually consistent
- More available (more 9s)
- Rolling updates
- 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 across versions 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)

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)

DISTRIBUTED SYSTEM GOALS TO CONSIDER

- Consider how the architecture may impact:
- Availability
- Accessibility
- Responsiveness
- Scalability
- Openness
- Distribution transparency
- Supporting resource sharing
- Other factors…

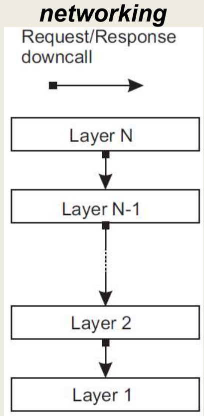
LAYERED ARCHITECTURES

- 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

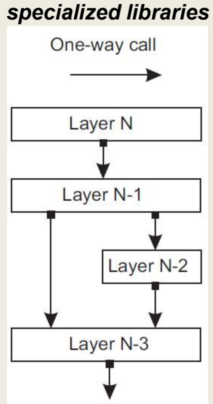
Pure-layered Organization

networking



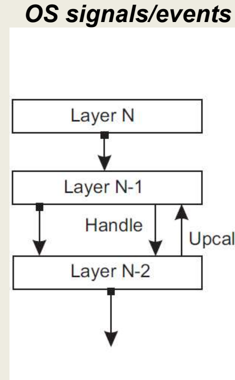
Mixed-layered organization

specialized libraries



Layered w/ upcalls organization

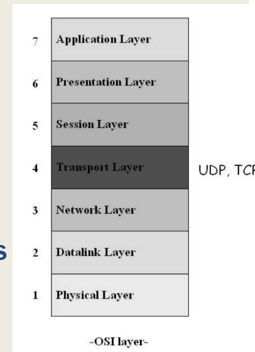
OS signals/events



COMMUNICATION-PROTOCOL STACKS

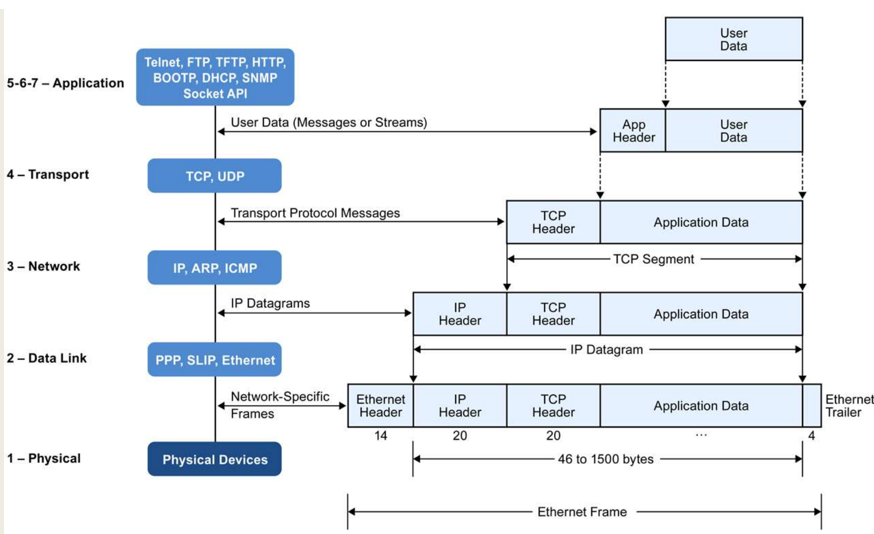
- Example: pure-layered organization
- Each layer offers an interface specifying functions of the layer
- Communication protocol: rules used for nodes to communicate
- Layer provides a <u>service</u>
- <u>Interface</u> makes service available
- <u>Protocol</u> implements communication for a layer

- New services can be built atop of existing layers to reuse low level implementation
- Abstractions make it easier reuse existing layers that already implement communication basics



January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.27

HOW A NETWORK PACKET IS BUILT

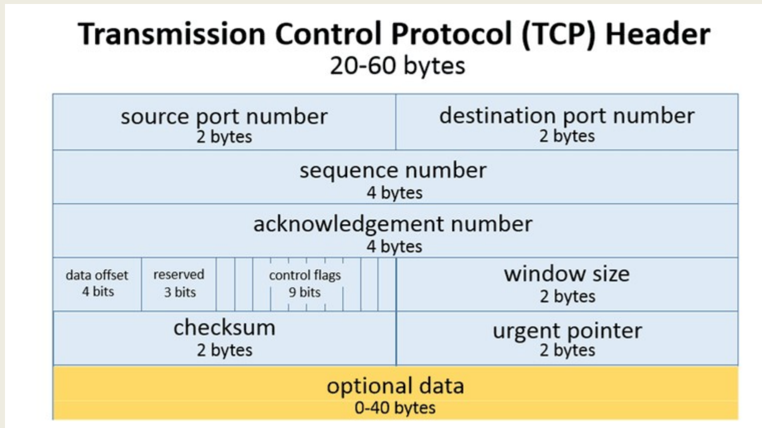


January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.28

TCP HEADER

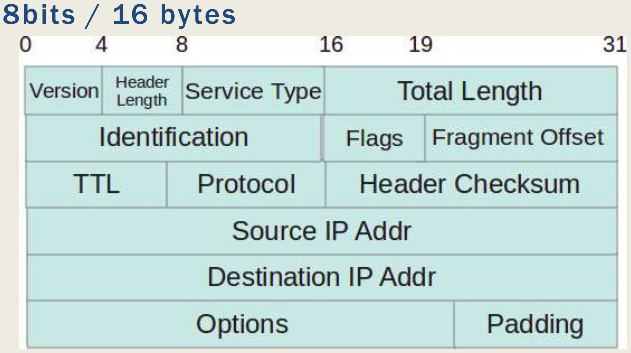
- Added in transport layer

Ports →
Pckt seq# →
Ackn # →



IP HEADER

- Added by network layer
- Source / Destination IP Address (no port)
- IPv4: 32bits / 4 bytes
- IPv6: 128bits / 16 bytes



TRANSMISSION CONTROL PROTOCOL (TCP)

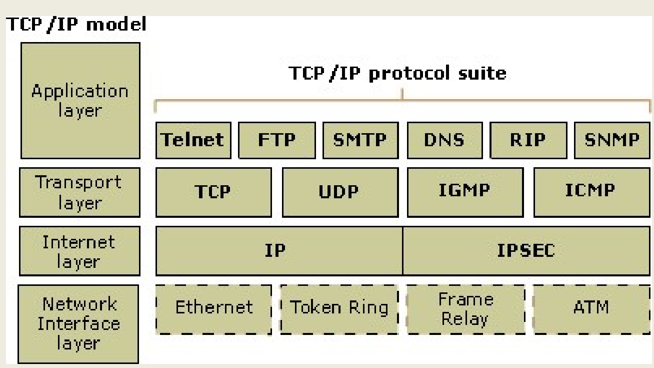
- TCP provides easy to use API
- API supports: setup, tear down of connection(s)
- API supports: sending and receiving of messages
- TCP preserves ordering of transferred data
- TCP detects and corrects lost data

- But TCP is “protocol” agnostic
  - E.g. language agnostic

- What are we going to say?
  - TCP does not dictate format or type/ordering of messages

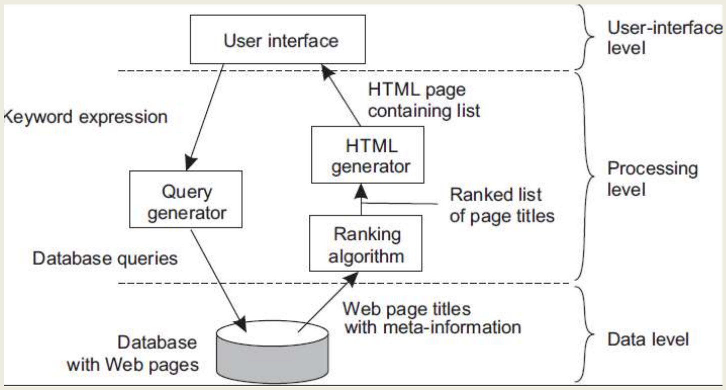
COMMON APPLICATION LAYER PROTOCOLS

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

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.35

ARCHITECTURAL STYLES

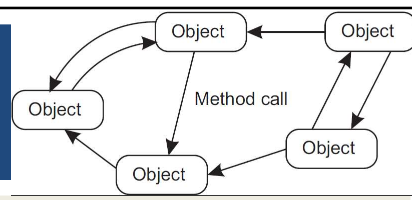
- Layered
- Object-based
  - Service oriented architecture (SOA)
- Resource-centered architectures
  - Representational state transfer (REST)
- Event-based
  - Publish and subscribe (Rich Site Summary RSS feeds)

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.36

OBJECT-BASED ARCHITECTURES

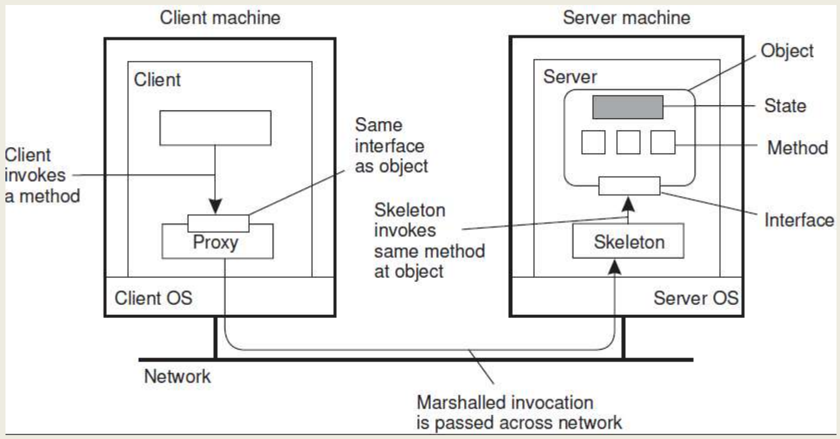
- Enables loose and flexible component organization
- Objects == components
- Enable distributed node interaction via function calls over the network
- Began with C - Remote Procedure Calls (RPC)
  - Straightforward: package up function inputs, send over network, transfer results back
  - Language independent
  - 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



DISTRIBUTED OBJECTS - 2

- 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

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)

RESOURCE BASED ARCHITECTURES

- Motivation:
  - Increasing number of services available online
  - Each with specific protocol(s), methods of interfacing
  - Connecting services w/ different protocols → integration nightmare
- Need for standardization of interfaces
  - Make services/components more pluggable
  - Easier to adopt and integrate
  - Common architecture

REST SERVICES

- Representational State Transfer (REST)
- Built on HTTP
- Four key characteristics:
  1. Resources identified through single naming scheme
  2. Services offer the same interface
     - Four operations: GET PUT POST DELETE
  3. Messages to/from a service are fully described
  4. After execution server forgets about client
     - Stateless execution

HYPERTEXT TRANSPORT PROTOCOL (HTTP)

- An ASCII-based request/reply protocol for transferring information on the web
- HTTP request includes:
  - request method (GET, POST, etc.)
  - Uniform Resource Identifier (URI)
  - HTTP protocol version understood by the client
  - headers—extra info regarding transfer request
- HTTP response from server
  - Protocol version & status code →
  - Response headers
  - Response body

HTTP status codes:
2xx — *all is well*
3xx — *resource moved*
4xx — *access problem*
5xx — *server error*

REST-FUL OPERATIONS

Operation	Description	
PUT	Create a new resource	(C)reate
GET	Retrieve state of a resource in some format	(R)ead
POST	Modify a resource by transferring a new state	(U)pdate
DELETE	Delete a resource	(D)elete

- Resources often implemented as objects in OO languages
- REST is weak for tracking state
- Generic REST interfaces enable ubiquitous *"so many"* clients

EXAMPLE: AMAZON S3

- Amazon S3 offers a REST-based interface
- Requires signing HTTP authorization header or passing authentication parameters in the URL query string
- REST: GET/PUT/POST/DELETE
- SOAP: 16 operations, moving toward deprecation
- Python boto ~50 operations (SDK for Python)
- SDKs for other languages

REST - 2

- Defacto web services protocol
- Requests made to a URI – uniform resource identifier
- Supersedes SOAP – Simple Object Access Protocol
- Access and manipulate web resources with a predefined set of stateless operations (known as web services)
- Responses most often in JSON, also HTML, ASCII text, XML, no real limits as long as text-based
- curl – generic command-line REST client: https://curl.haxx.se/


```
// WSDL Service Definition
<?xml version="1.0" encoding="UTF-8"?>
<definitions  name ="DayOfWeek"
  targetNamespace="http://www.roguewave.com/soapworx/examples/DayOfWeek.wsdl"
  xmlns:tns="http://www.roguewave.com/soapworx/examples/DayOfWeek.wsdl"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="DayOfWeekInput">
    <part name="date" type="xsd:date"/>
  </message>
  <message name="DayOfWeekResponse">
    <part name="dayOfWeek" type="xsd:string"/>
  </message>
  <portType name="DayOfWeekPortType">
    <operation name="GetDayOfWeek">
      <input message="tns:DayOfWeekInput"/>
      <output message="tns:DayOfWeekResponse"/>
    </operation>
  </portType>
  <binding name="DayOfWeekBinding" type="tns:DayOfWeekPortType">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="GetDayOfWeek">
      <soap:operation soapAction="getdayofweek"/>
      <input>
        <soap:body use="encoded"
          namespace="http://www.roguewave.com/soapworx/examples"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body use="encoded"
        namespace="http://www.roguewave.com/soapworx/examples"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>
  <service name="DayOfWeekService" >
    <documentation>
      Returns the day-of-week name for a given date
    </documentation>
    <port name="DayOfWeekPort" binding="tns:DayOfWeekBinding">
      <soap:address location="http://localhost:8090/dayofweek/DayOfWeek"/>
    </port>
   </service>
</definitions>
```

```
// REST/JSON
// Request climate data for Washington

{
 "parameter": [
  {
    "name": "latitude",
    "value":47.2529
  },
  {
    "name": "longitude",
    "value":-122.4443
  }
  ]
}
```

L4.51

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)

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.52

PUBLISH-SUBSCRIBE ARCHITECTURES

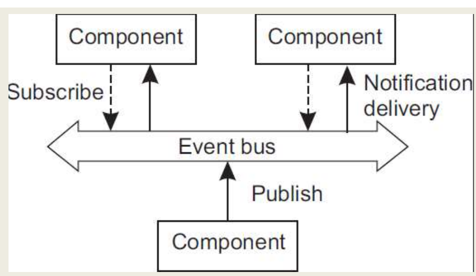
- Enables separation between processing and coordination
- Types of coordination:

	Temporally coupled (at the same time)	Temporally decoupled (at different times)
Referentially coupled (*dependent on name*)	**Direct** Explicit synchronous service call	**Mailbox** Asynchronous by name (address)
Referentially decoupled (*name not required*)	**Event-based** Event notices published to shared bus, w/o addressing	**Shared data space** Processes write tuples to a shared data space

Not publish and subscribe

October 12, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L5.53

PUBLISH-SUBSCRIBE ARCHITECTURES - 2

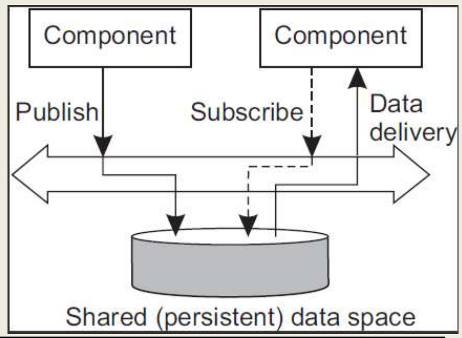


- **Event-based coordination**
- Processes do not know about each other explicitly
- Processes:
  - **Publish:** a notification describing an event
  - **Subscribe:** to receive notification of specific kinds of events
- Assumes subscriber is presently up (*temporally coupled*)

October 12, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L5.54

PUBLISH SUBSCRIBE ARCHITECTURES - 3

- **Shared data space**
- Full decoupling (name and time)
- Processes publish "tuples" to shared dataspace (publish)
- Processes provide search pattern to find tuples (subscribe)
- When tuples are added, subscribers are notified of matches
- **Key characteristic:** Processes have no explicit reference to each other



PUBLISH SUBSCRIBE ARCHITECTURES - 4

- Subscriber describes events interested in
- Complex descriptions are intensive to evaluate and fulfil
- **Middleware will:**
- Publish matching notification and data to subscribers
  - Common if middleware lacks storage
- Publish only matching notification
  - Common if middleware provides storage facility
  - Client must explicitly fetch data on their own
- Publish and subscribe systems are generally scalable
- **What would reduce the scalability of a publish-and-subscribe system?**

RESEARCH DIRECTIONS

THIS WINTER

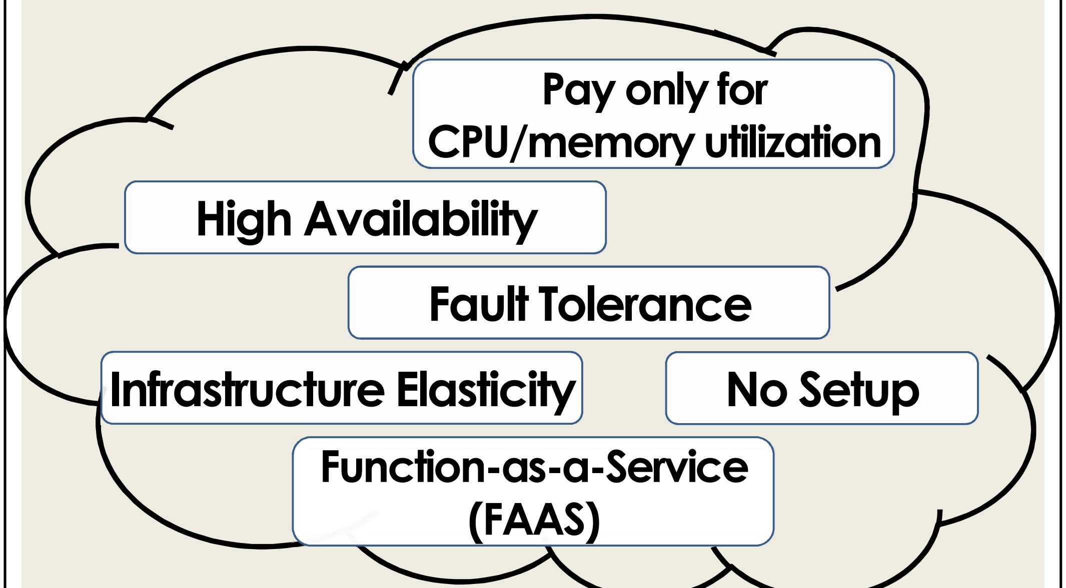
- Research group meetings
  - Cloud/Distributed Sys - Tuesdays 12:00-1:30pm, MDS 312
  - Bioinformatics – Wednesday 11:30-1:00pm, TLB 307C
- **Goals**:
  - Assemble ongoing agile research teams which maximize opportunities for student collaboration and sharing to lower the bar for student engagement 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/

59

SERVERLESS COMPUTING



SERVERLESS COMPUTING

Why Serverless Computing?

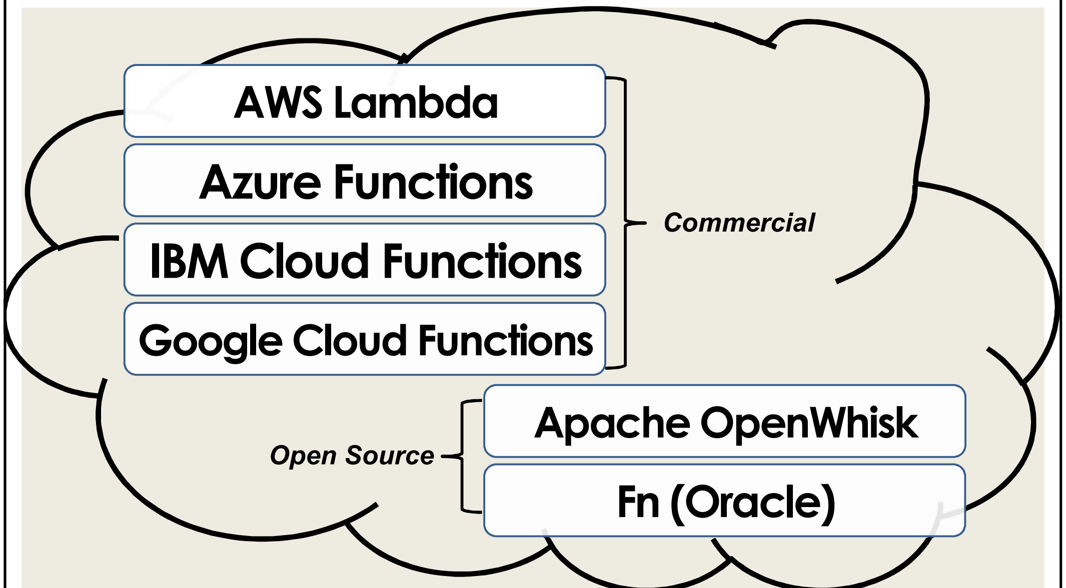
Many features of distributed systems, that are challenging to deliver, are provided automatically

…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



SERVERLESS COMPUTING

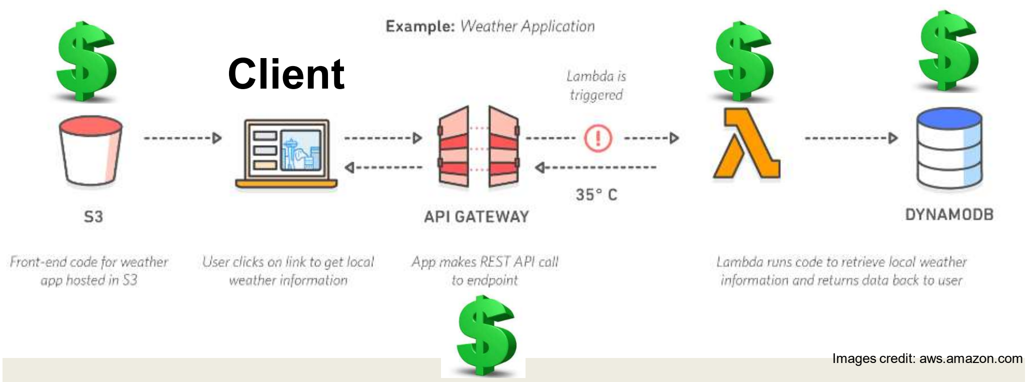
Research Challenges

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

64

VENDOR ARCHITECTURAL LOCK-IN

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



Images credit: aws.amazon.com

- 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

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	66

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

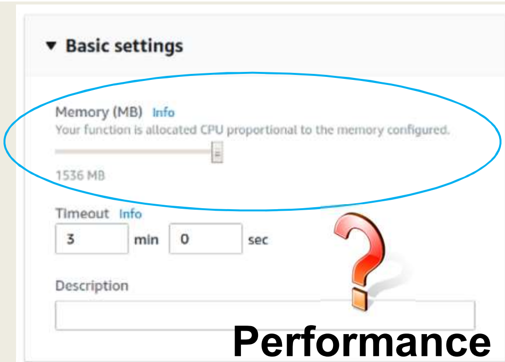
Break Even: 4,319,136 GB-sec

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…

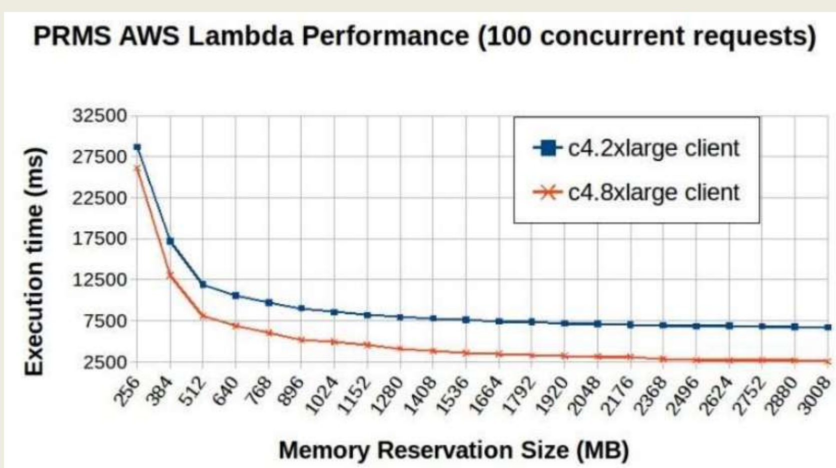
- Lambda memory reserved for functions
- UI provides "slider bar" to set function's memory allocation
- Resource capacity (CPU, disk, network) coupled to slider bar: *"every **doubling** of memory, **doubles** CPU…"*



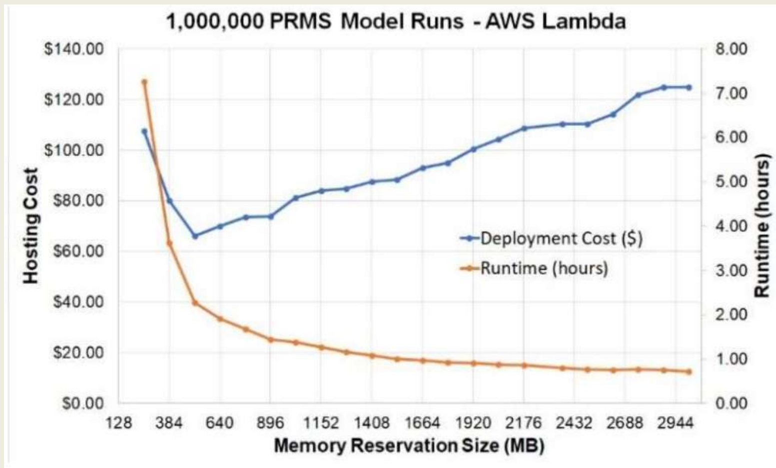
- But how much memory do model services require?

LAMBDA: PERFORMANCE VS MEMORY

- Order of magnitude performance gain ~ 10x



HOW MUCH FOR 1,000,000 CALLS?



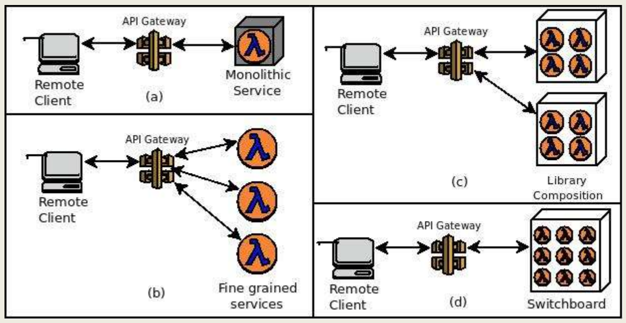
CLOUD NATIVE APPLICATIONS: EVOLVING BEST PRACTICES

- 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

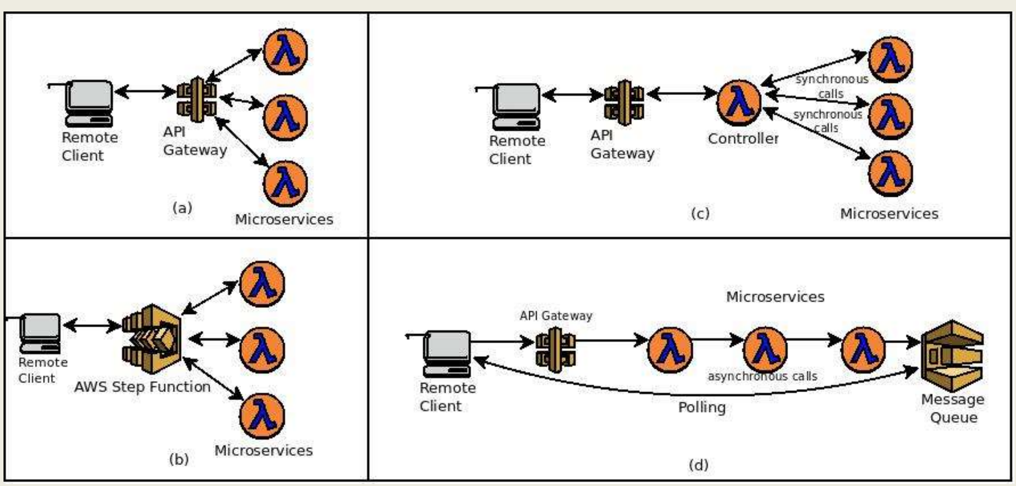
- How should application code be composed for deployment to FaaS platforms?

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

Performance

- 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

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

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	76

RESEARCH DIRECTIONS

October 5, 2017 TCSS558: Applied Distributed Computing [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma L10.77

SERVERLESS COMPUTING

- ***FaaS Inspector Project*** – *Multi.Students, Shruti Ramesh (Microsoft)*
  https://github.com/wlloyduw/faas_inspector
- ***Service composition*** – *Baojia Zhang*
  - *Performance and cost implications of microservice disaggregation vs. composition*
- FaaS Performance Simulation and Modeling – *Lan Ly*
- Freeze/Thaw Lifecycle Mitigation – *Minh Vu*
- Cloud vs Edge vs Device – *Harrison Ross*
- Unique applications of FaaS:
  - Computer Vision Neural Networks – *Vlad Kaganyuk (t-mobile)*
  - Gaming, Bioinformatics, others…
  - FaaS Application Migration – *Baojia Zhang*

CONTAINERIZATION

- Application system containers - Docker
- Container orchestration framework(s) – Kubernetes, Docker Swarm, Apache Mesos/Marathon, AWS Elastic Container Service
- <u>*Container-as-a-Service*</u> – "Serverless" alternative to container orchestration frameworks, looking for student to conduct MSCSS project to explore this new technology (AWS Fargate, Azure Container Instances, Google…)
- 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*
  - Tsunami Modeling (on AWS GPU instances) – *Shawn Qin*
- 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
  - <u>Public cloud resource contention and avoidance</u> – *Edward Han, Jugal Gandhi*

VIRTUALIZATION / UNIKERNELS

- Lightweight alternative to containers and VMs
  - Custom Cloud Operating System
  - No/one process, multiple threads, run one program
  - Launch separately atop of hypervisor (XEN/KVM)
  - Reduce overhead, duplication of heavy weight OS

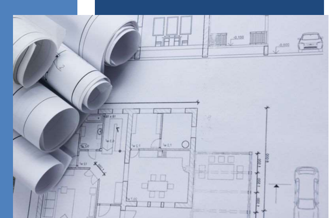
  - Performance comparison to containers, virtual machines
  - Web application (services) and native Java application comparison (OSv) - *Devin Durham*
  - Comparison study: unikernels vs. containers vs. VMs
  - ***(NEW!)*** Micro VMs: AWS Firecracker https://github.com/firecracker-microvm/firecracker

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? (SERVERLESS FAAS / IAAS)
  - 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***



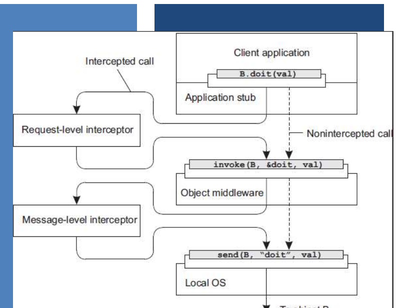
IN-CLASS ACTIVITY: DISTRIBUTED SYSTEMS ARCHITECTURES

L5.84

DISTRIBUTED SYSTEM GOALS TO CONSIDER

- Consider how the architectural change may impact:
- Availability
- Accessibility
- Responsiveness
- Scalability
- Openness
- Distribution transparency
- Supporting resource sharing
- Other factors…

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.85



MIDDLEWARE ORGANIZATION

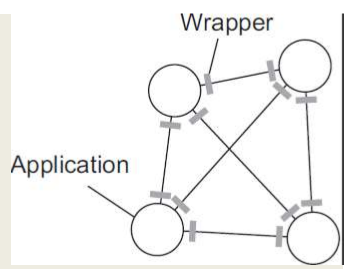
October 12, 2017 | TCSS558: Applied Distributed Computing [Winter 2019]
School of Engineering and Technology, University of Washington - Tacoma | L10.86

MIDDLEWARE: WRAPPERS

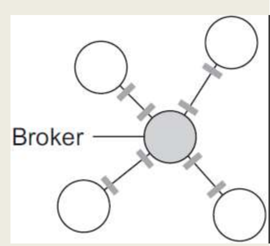
- **Wrappers (adapters)**
  - Special "frontend" components that provide interfaces to client
  - Interface wrappers transform client requests to "implementation" at the component-level
  - Provide modern services interfaces for legacy code/systems
  - Enable meeting all preconditions for legacy code to operate
  - Parameterization of functions, configuration of environment
- Contributes towards system openness
- **Example: Amazon S3**
- Client uses REST interface to GET/PUT/DELETE/POST data
- S3 adapts and hands off REST requests to system for fulfillment

MIDDLEWARE: WRAPPERS - 2

- Inter-application communication
  - Application provides unique interface for every application
- Scalability suffers
  - N applications → $O(N^2)$ wrappers



- **Broker**
  - Provide a common intermediary
  - Broker knows how to communicate with every application
  - Applications only know how to communicate with the broker



MIDDLEWARE: INTERCEPTORS

- **Interceptor**
- Software construct, breaks flow of control, allows other application code to be executed
- Enables remote procedure calls (RPC), remote method invocation (RMI)
- Object A can call a method belonging to object B on a different machine than A.

October 12, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L5.89

MIDDLEWARE INTERCEPTION - METHOD

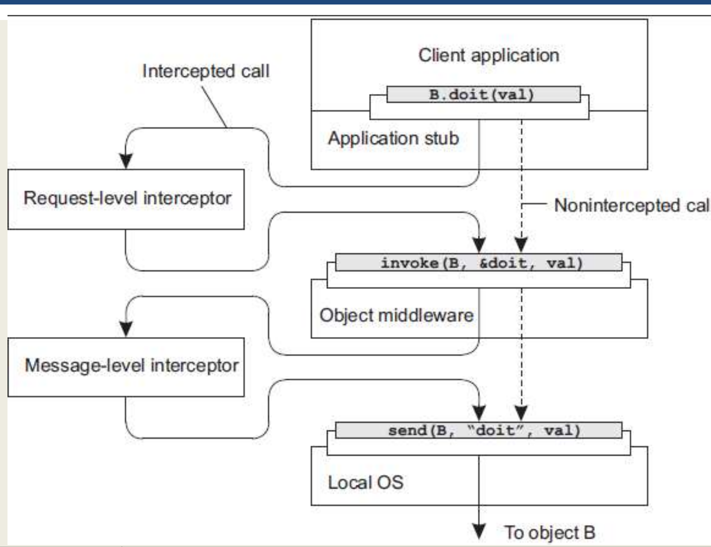
- Local interface matching Object B is provided to Object A
- Object A calls method in this interface
- A's call is transformed into a "generic object invocation" by the middleware
- The "generic object invocation" is transformed into a **message** that is sent over Object A's network to Object B.
- Request-level interceptor automatically routes all calls to object replicas

October 12, 2017	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L5.90

MODIFIABLE MIDDLEWARE

- It should be possible to modify middleware without loss of availability
- Software components can be replaced at runtime
- Component-based design
  - Modifiability through composition
  - Systems may have static or dynamic configuration of components
  - Dynamic configuration requires ***late binding***
  - Components can be changed at runtime
- Component based software supports modifiability at runtime by enabling components to be swapped out.
- **Does a microservices architecture (e.g. AWS Lambda) support modifiability at runtime ?**

MIDDLEWARE: INTERCEPTORS - 2



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

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.95

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

January 16, 2019	TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma	L4.96

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)

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?