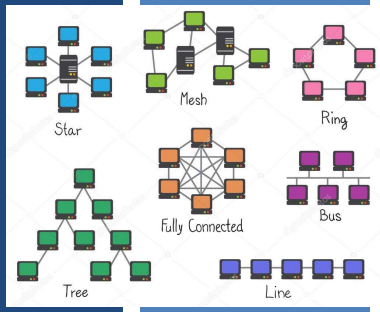
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Distributed Systems Architectures



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

FEEDBACK – 1/23

- **I'm confused about *accessibility* vs. *availability* in distributed systems…**
- ***Accessibility***: refers to making remote resources (e.g. servers, storage, networks, data) easy for users to access
  - In cloud computing, each service delivery model (e.g. IAAS, PAAS, FAAS) provides accessibility through a different interface / API
  - Can evaluate which interface(s) are easier to use…
- ***Availability***: refers to making remote resources available around-the-clock
  - Ranked using 9s: 99%, 99.9%, 99.99%
  - High availability (HA) - systems designed with fail-over HW to "always" be available
  - HA systems feature fault tolerance from HW failures

FEEDBACK – 2

- **What is an example of a system having a shared data architecture? (e.g. shared data space model)**
- Systems that feature referential (name) and temporal (time) decoupling
- Distributed systems where nodes communicate with messaging middleware
- Common messaging middleware: RabbitMQ, Apache Kafka, AWS SQS
- Messaging middleware supports referential and temporal decoupling through a publish and subscribe pattern
- Publishers submit messages to message queues
- Subscribers later retrieve messages from queues, or receive notifications of message availability

FEEDBACK - 3

- **How do clients in event-based architectures access the events when there is no name/reference?**
- ***In other words, how do we know to subscribe and consume event-based data --- in contrast to using REST APIs, where clients invoke REST services***
- System will consist of distributed nodes
- Nodes connect to event bus or message queue on start up
- Difference from shared data space is that nodes must be active and online as messages only are disseminated once in response to events
- Nodes that go offline miss messages
  - When nodes come back online, messages are not persisted

FEEDBACK - 4

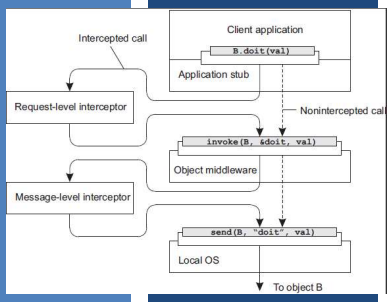
- **Will we have to know TCP handshake (Syn/Ack) details?**
- When implementing application protocols on top of TCP *(assignment 1)*, details are hidden in lower OSI layers, so intimate knowledge is generally not required

FEEDBACK - 5

- **What does windowing mean for TCP communication?**
- Sender and Receiver use sliding buffers to communicate
- Idea is to acknowledge receipt of portion of message
- Example:
- Two hosts A and B each allocate 32KB buffers for incoming data
- Host A sends data to Host B; B advertises 32,768 byte window size
- Host A understands it can send 32,768 bytes before receiving any acknowledgement from Host B
- Given a message segment size (MSS) of 1,460 bytes, 22 segments can be sent without acknowledgement
- When Host B acknowledges receipt, can advertise a smaller window size (e.g. 16 KB) if still an application is still processing the data

OBJECTIVES

- Homework 0 Questions
- Homework 1 posted…
- Chapter 2: System architectures
  - Centralized: Single client, multi-tier
  - Decentralized peer-to-peer: structured, unstructured, hierarchical
  - Hybrid
- Chapter 3 Processes
  - 3.1 Threads



SYSTEM ARCHITECTURES

SYSTEM ARCHITECTURES

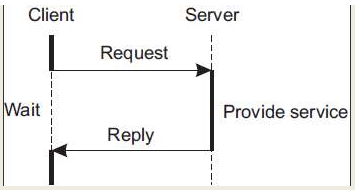
- Architectural styles (or patterns)
- General, reusable solutions to commonly occurring system design problems
- Expressed as a logical organization of components and connectors
- Deciding on the system components, their interactions, and placement is a realization of a **system architecture**
- System architectures represent designs used in practice

TYPES OF SYSTEM ARCHITECTURES

- Centralized system architectures
  - Client-server
  - Multitiered
- Decentralized peer-to-peer architectures
  - Structured
  - Unstructured
  - Hierarchically organized
- Hybrid architectures

CENTRALIZED: SIMPLE CLIENT-SERVER ARCHITECTURE

- **Clients** request services
- **Servers** provide services
- Request-reply behavior



- **Connectionless protocols (UDP)**
- Assume stable network communication with no failures
- Best effort communication: No guarantee of message arrival without errors, duplication, delays, or in sequence. No acknowledgment of arrival or retransmission
- Problem: How to detect whether the client request message is lost, or the server reply transmission has failed
- Clients can resend the request when no reply is received
- ***But what is the server doing?***

CLIENT-SERVER PROTOCOLS

- **Connectionless cont'd**
- Is resending the client request a good idea?
- Examples:
  Client message: "transfer $10,000 from my bank account"
  Client message: "tell me how much money I have left"
- **Idempotent** – repeating requests is safe
- **Connection-oriented (TCP)**
- Client/server communication over wide-area networks (WANs)
- When communication is inherently reliable
- Leverage "reliable" TCP/IP connections

CLIENT-SERVER PROTOCOLS - 2

- **Connection-oriented cont'd**
- Set up and tear down of connections is relatively expensive
- Overhead can be amortized with longer lived connections
  - Example: database connections often retained
- Ongoing debate:
- How do you differentiate between a client and server?
- Roles are ***blurred***
- **Blurred Roles Example:** Distributed databases
- DB nodes both **service** client requests, *and* ***submit*** new requests to other DB nodes for replication, synchronization, *etc.*

TCP/UDP

TCP	UDP
Reliable	Unreliable
Connection-oriented	Connectionless
Segment retransmission and flow control through windowing	No windowing or retransmission
Segment sequencing	No sequencing
Acknowledge segments	No acknowledgement

CONNECTIONLESS VS CONNECTION ORIENTED

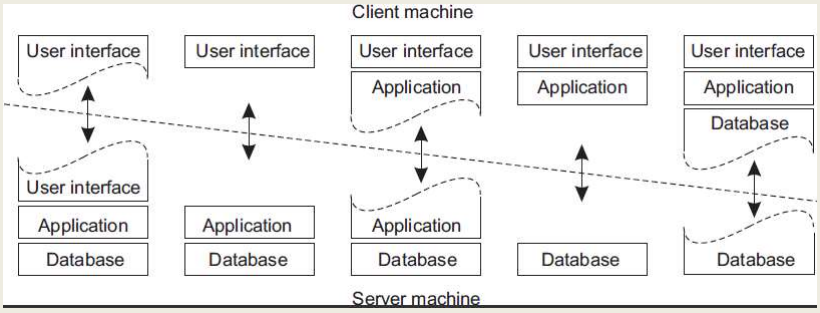
	Connectionless (UDP) *stateless*	Connection-oriented (TCP) *stateful*
Advantages		
Disadvantages		

CONNECTIONLESS VS CONNECTION ORIENTED

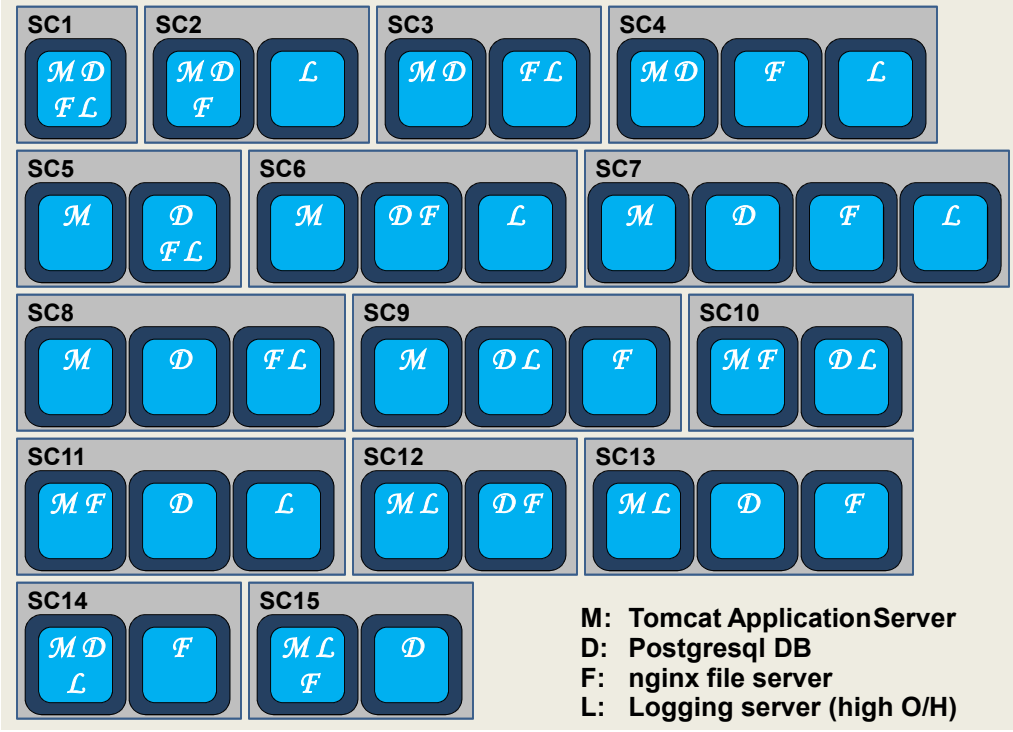
	Connectionless (UDP) *stateless*	Connection-oriented (TCP) *stateful*
Advantages	• Fast to communicate (no connection overhead) • Broadcast to an audience • Network bandwidth savings	• Message delivery confirmation • Idempotence not required • Messages automatically resent - if client (or network) is temporarily unavailable • Message sequences guaranteed
Disadvantages	• Cannot tell difference of request vs. response failure • Requires idempotence • Clients must be online and ready to receive messages	• Connection setup is time-consuming • More bandwidth is required (protocol, retries, multinode-communication)

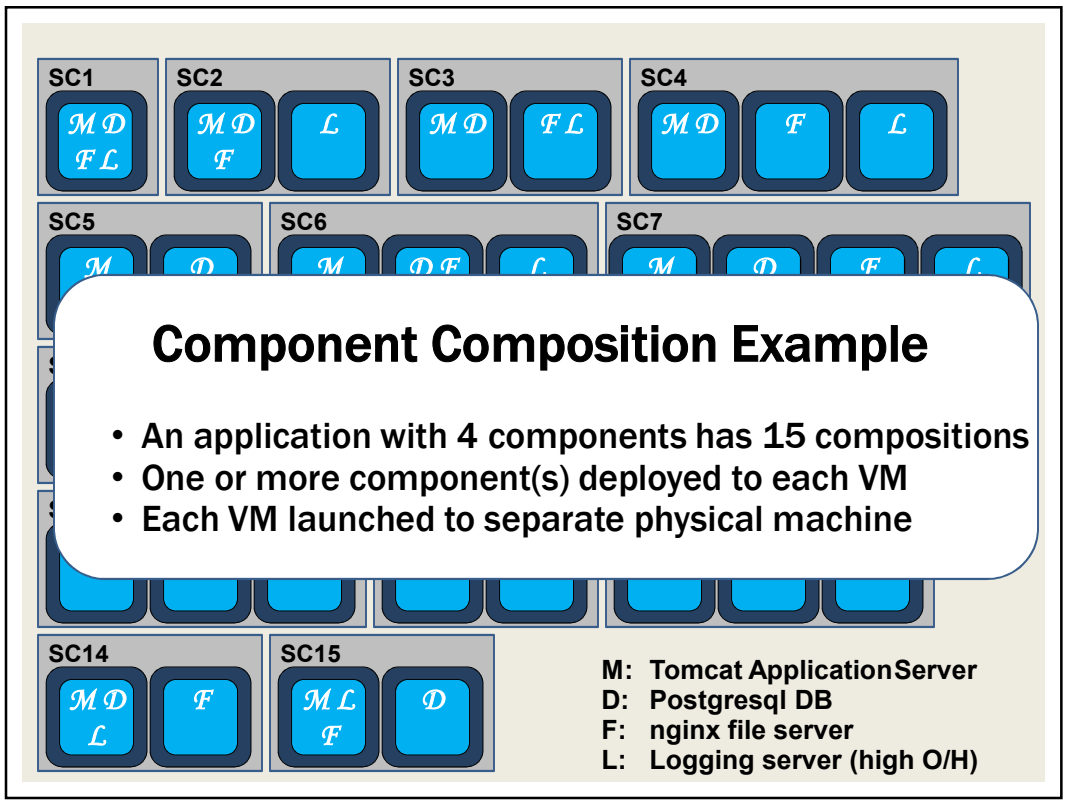
MULTITIERED ARCHITECTURES

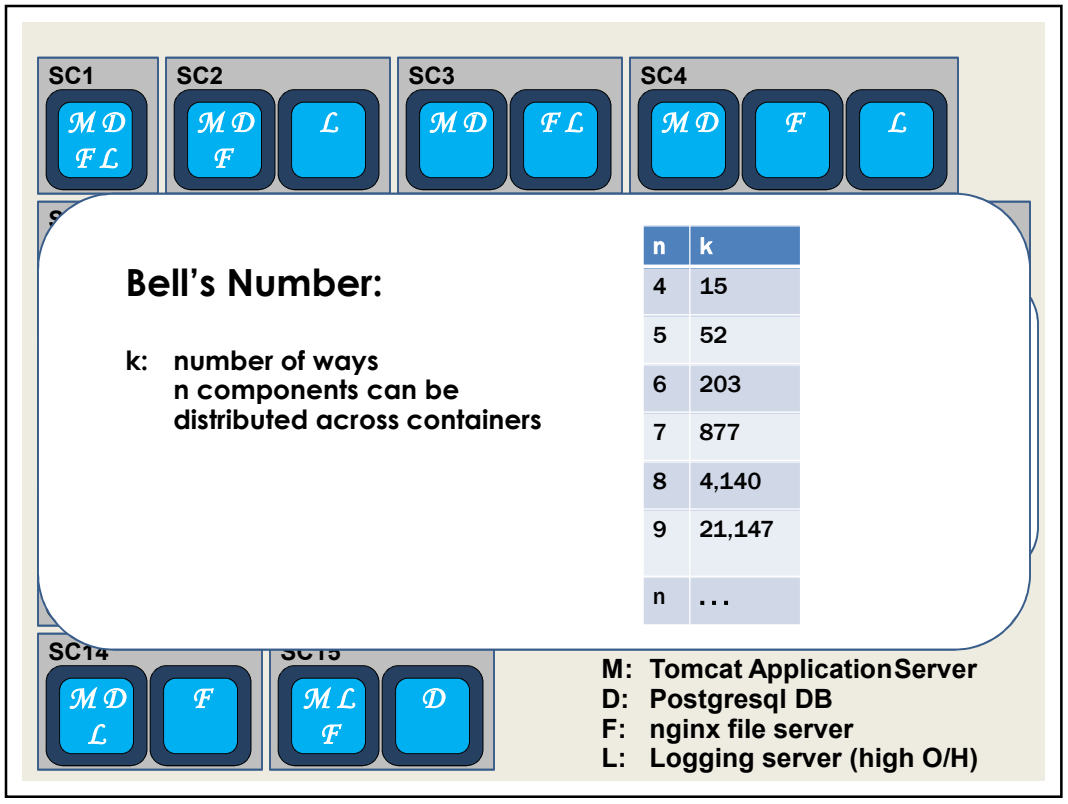
- Where should functionality be distributed?
  - At the client?
  - At the server?

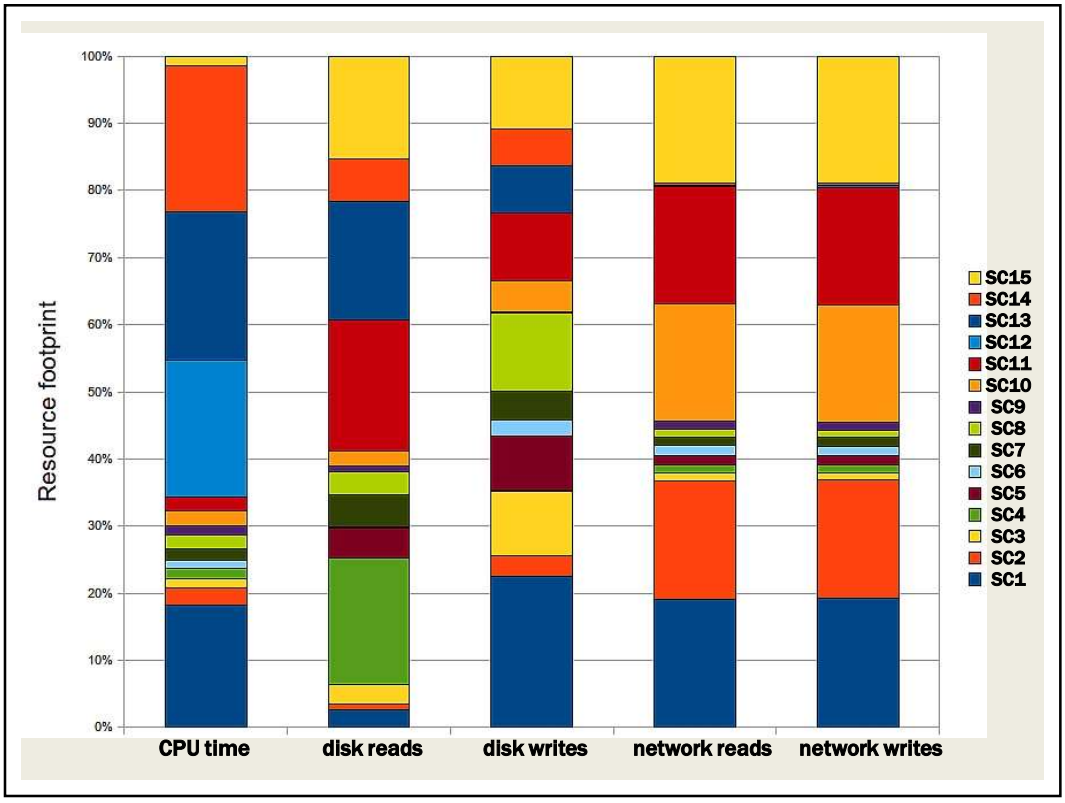


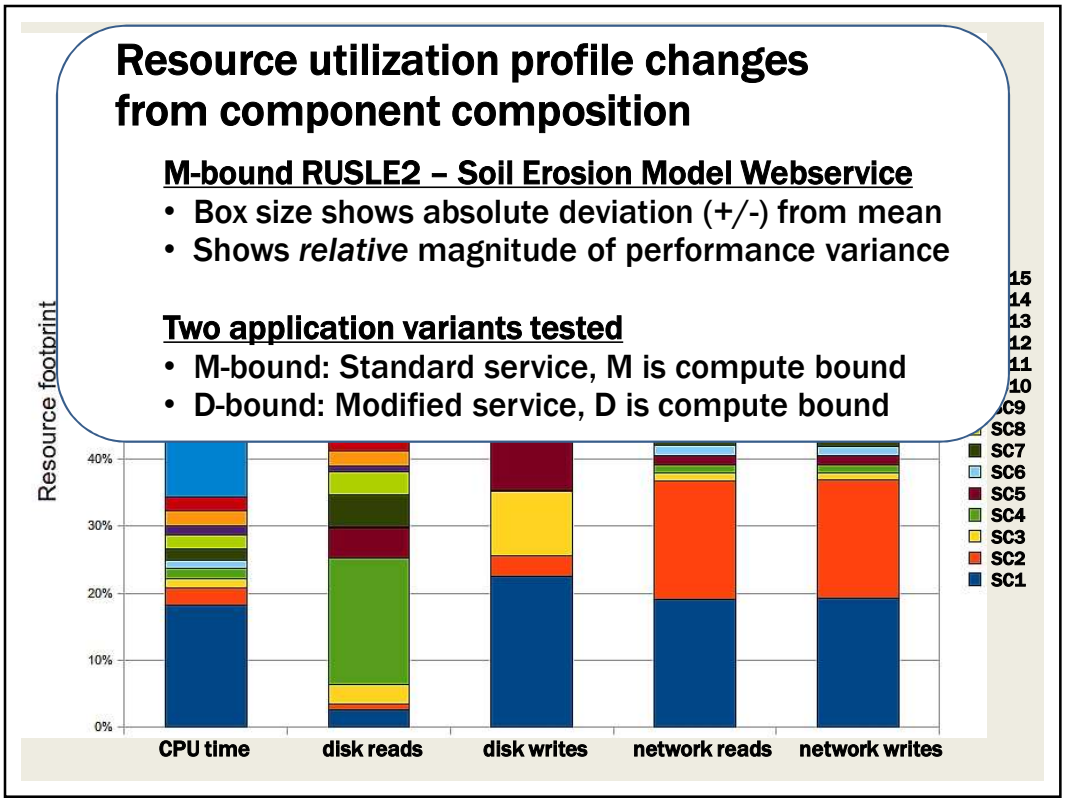
- Why should we consider component composition?

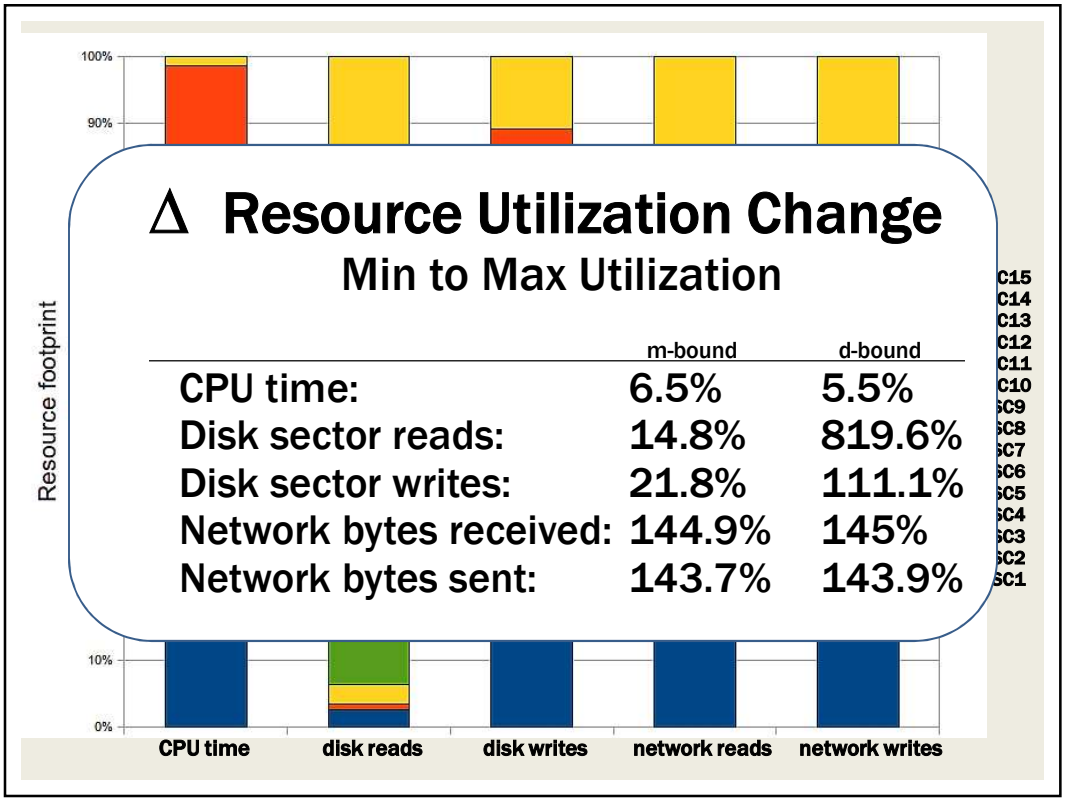



SC1
SC2
SC3
SC4
SC5
SC6
SC7
SC14
SC15
Component Composition Example
• An application with 4 components has 15 compositions
• One or more component(s) deployed to each VM
• Each VM launched to separate physical machine
M: Tomcat ApplicationServer
D: Postgresql DB
F: nginx file server
L: Logging server (high O/H)


SC1
SC2
SC3
SC4
SC14
Bell's Number:
k: number of ways n components can be distributed across containers
n | k
4 | 15
5 | 52
6 | 203
7 | 877
8 | 4,140
9 | 21,147
n | . . .
M: Tomcat ApplicationServer
D: Postgresql DB
F: nginx file server
L: Logging server (high O/H)


Resource footprint
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
CPU time
disk reads
disk writes
network reads
network writes
SC15
SC14
SC13
SC12
SC11
SC10
SC9
SC8
SC7
SC6
SC5
SC4
SC3
SC2
SC1


Resource utilization profile changes from component composition
M-bound RUSLE2 – Soil Erosion Model Webservice
• Box size shows absolute deviation (+/-) from mean
• Shows *relative* magnitude of performance variance
Two application variants tested
• M-bound: Standard service, M is compute bound
• D-bound: Modified service, D is compute bound
Resource footprint
40%
30%
20%
10%
0%
CPU time
disk reads
disk writes
network reads
network writes
SC8
SC7
SC6
SC5
SC4
SC3
SC2
SC1



Δ Resource Utilization Change

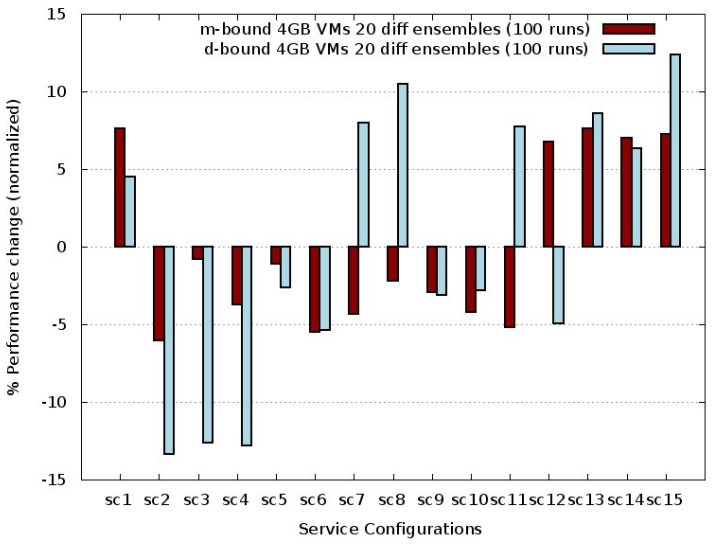
Min to Max Utilization

	m-bound	d-bound
CPU time:	6.5%	5.5%
Disk sector reads:	14.8%	819.6%
Disk sector writes:	21.8%	111.1%
Network bytes received:	144.9%	145%
Network bytes sent:	143.7%	143.9%

PERFORMANCE IMPLICATIONS OF COMPONENT DEPLOYMENTS

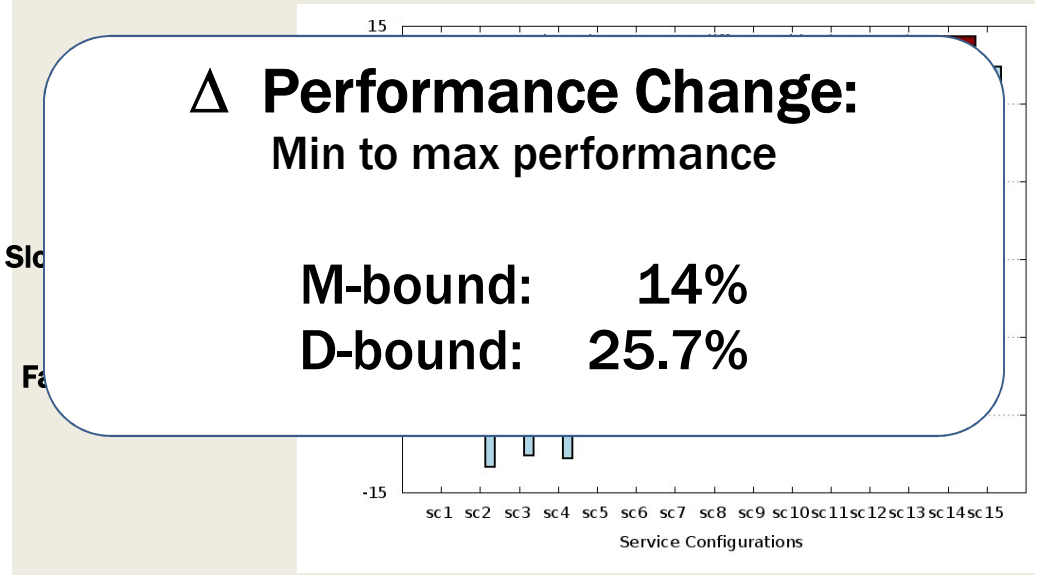
Slower deployments

Faster deployments



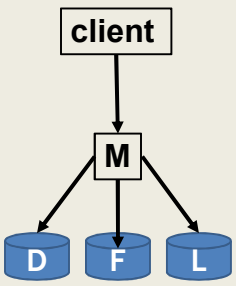
24

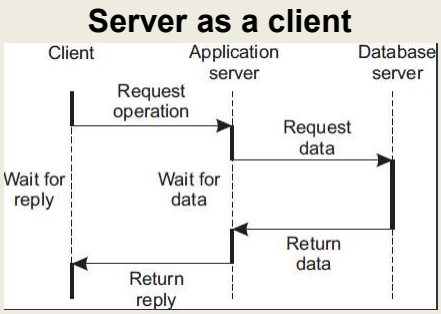
PERFORMANCE IMPLICATIONS OF COMPONENT DEPLOYMENTS



MULTITIERED ARCHITECTURES - 2

- M D F L architecture
- M – is the application server
- M – is also a client to the database (D), fileserver (F), and logging server (L)





MULTITIERED RESOURCE SCALING

- **Vertical distribution**
- The distribution of "M D F L"
- Application is scaled by placing "tiers" on separate servers
  - M – The application server
  - D – The database server
- Vertical distribution impacts "network footprint" of application
- Service isolation: each component is isolated on its own HW

- **Horizontal distribution**

- Scaling an individual tier
- Add multiple machines and distribute load
- Load balancing (with a load balancer *haproxy, nginx)*

MULTITIERED RESOURCE SCALING - 2

- **Horizontal distribution cont'd**
  - Sharding: portions of a database map to a specific server
  - A distributed hash table can be used to resolve which server holds the data
  - Relational databases can also be scaled horizontally by having replica servers

TYPES OF SYSTEM ARCHITECTURES

- Centralized system architectures
  - Client-server
  - Multitiered
- Decentralized peer-to-peer architectures
  - Structured
  - Unstructured
  - Hierarchically organized
- Hybrid architectures

DECENTRALIZED PEER-TO-PEER ARCHITECTURES

- Client/server:
  - Nodes have specific roles
- Peer-to-peer:
  - Nodes are seen as ***all equal…***
- **How should nodes be organized for communication?**

STRUCTURED PEER-TO-PEER

- Nodes organized using specific ***topology*** (e.g. ring, binary-tree, grid, etc.)
  - Organization (structure) assists in data lookups
- Data indexed using "semantic-free" indexing
  - Key / value storage systems
  - Key used to look-up data
- Nodes store data associated with a subset of keys

DISTRIBUTED HASH TABLE (DHT)

- Distributed hash table (DHT) *(ch. 5)*
- Hash function

```
key(data item) = hash(data item's value)
```

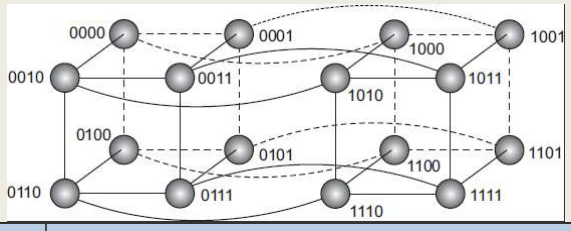
- Hash function "generates" a unique key based on the data
- No two data elements will have the same key (hash)
- System supports data lookup via key
- ***Any*** node can receive and resolve the request
- Lookup function determines which node stores the key

```
existing node = lookup(key)
```

- Node forwards request to node with the data
- ***DOES this approach provide distribution transparency to clients?***

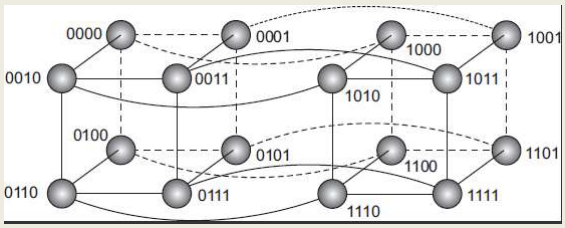
FIXED HYPERCUBE EXAMPLE

- Example where topology helps ***route*** data lookup request
- Statically sized 4-D hypercube, every node has 4 connectors
- 2 x 3-D cubes, 8 vertices, 12 edges
- Node IDs represented as 4-bit code (0000 to 1111)
- Hash data items to 4-bit key (1 of 16 slots)
- Distance (number of hops) determined by identifying number of varying bits between neighboring nodes and destination



FIXED HYPERCUBE EXAMPLE - 2

- **Example:** *fixed hypercube*
  node 0111 (7) retrieves data from node 1110 (14)
- Node 1110 is not a neighbor to 0111
- **Which connector leads to the shortest path?**



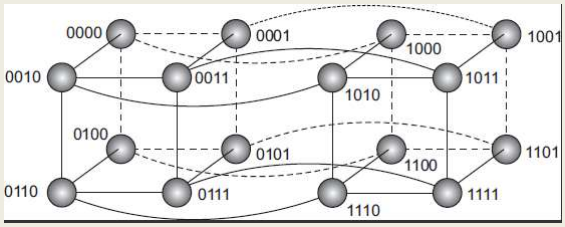
WHICH CONNECTOR LEADS TO THE SHORTEST PATH?

- **Example:** node 0111 (7) retrieves data from node 1110 (14)
- Node 1110 is not a neighbor to 0111

[0111] Neighbors:

1111 (1 bit different than 1110) 0011 (3 bits different– bad path)

0110 (1 bit different than 1110) 0101 (3 bits different– bad path)



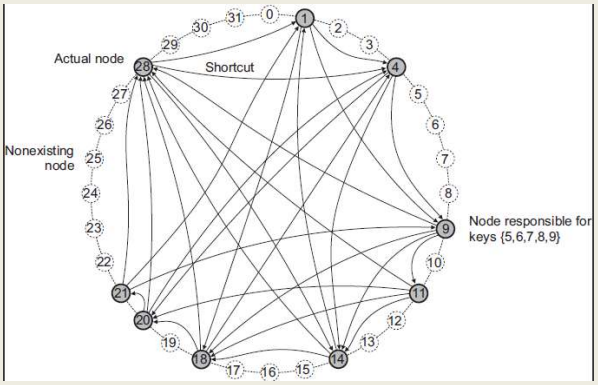
DYNAMIC TOPOLOGY

- Fixed hypercube requires **static topology**
  - Nodes cannot join or leave → *what if 1 node short of perfect cube?*
- Relies on symmetry of number of nodes
- Can force the DHT to a certain size

- Chord system – DHT (in ch.5)
  - Dynamic topology
  - Nodes organized in ring
  - Every node has unique ID
  - Each node connected with other nodes (shortcuts)
  - Shortest path between any pair of nodes is ~ order O(log N)
  - N is the total number of nodes

CHORD SYSTEM

- Data items have m-bit key
- Data item is stored at closest "successor" node with ID ≥ key k
- Each node maintains finger table of successor nodes
- Client sends key/value lookup to ***any*** node
- Node forwards client request to node with m-bit ID closest to, but not greater than key k
- Nodes must continually refresh finger tables by communicating with adjacent nodes to incorporate node joins/departures



UNSTRUCTURED PEER-TO-PEER

- **No topology:** ***How do nodes find out about each other?***
- Each node maintains ad hoc list of neighbors
- Facilitates nodes frequently joining, leaving, ad hoc systems
- **Neighbor:** node reachable from another via a network path
- Neighbor lists constantly refreshed
  - Nodes query each other, remove unresponsive neighbors
- Forms a "random graph"
- Predetermining network routes not possible
  - How would you calculate the route algorithmically?
- Routes must be discovered

UNSTRUCTURED PEER-TO-PEER

- Methods to find/disseminate data in unstructured peer-to-peer networks
- Flooding
- Random Walks
- Policy-based search
- Alternate topology:
- Hierarchically organized peer-to-peer networks

SEARCHING FOR DATA: UNSTRUCTURED PEER-TO-PEER SYSTEMS

- **Flooding**
- `[Node u]` sends request for data item to all neighbors
- `[Node v]`
  - Searches locally, responds to `[Node u]` (or forwarder) if having data
  - Forwards request to **ALL** neighbors
  - Ignores repeated requests
- Features
  - High network traffic
  - Fast search results by saturating the network with requests
  - Variable # of hops
  - Max number of hops or time-to-live (TTL) often specified
  - Requests can "retry" by gradually increasing TTL/max hops until data is found

SEARCHING FOR DATA - 2

- **Random walks**
- [Node u] asks a randomly chosen neighbor [node v]
- If [node v] does not have data, forwards request to a random neighbor
- Features
  - Low network traffic
  - Akin to sequential search
  - Longer search time
  - [node u] can perform parallel random walks to reduce search time
  - As few as 16..64 random walks effective to reduce search time
  - Timeout required - need to coordinate stopping network-wide walk when data is found…

SEARCHING FOR DATA - 3

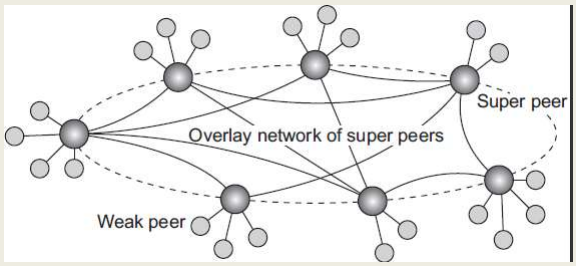
- **Policy-based search methods**
- Incorporate history and knowledge about the ad hoc network ***at the node-level*** to enhance effectiveness of queries
- Nodes maintain lists of preferred neighbors which often succeed at resolving queries
- Favor neighbors having highest number of neighbors
  - Can help minimize hops

HIERARCHICALLY ORGANIZED PEER-TO-PEER NETWORKS

- **Problem:**
  Ad hoc system search performance does not scale well as system grows
- Allow nodes to assume roles to improve search
- Content delivery networks (CDNs) *(video streaming)*
  - Store (cache) data at nodes local to the requester (client)
  - Broker node – tracks resource usage and node availability
    - Track where data is needed
    - Track which nodes have capacity (disk/CPU resources) to host data
- Node roles
  - Super peer –Broker node, routes client requests to storage nodes
  - Weak peer – Store data

HIERARCHICALLY ORGANIZED PEER-TO-PEER NETWORKS - 2

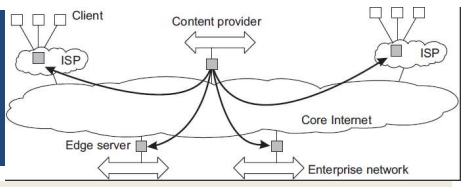
- Super peers
  - Head node of local centralized network
  - Interconnected via overlay network with other super peers
  - May have replicas for fault tolerance
- Weak peers
  - Rely on super peers to find data
- Leader-election problem:
  - Who can become a super peer?
  - What requirements must be met to become a super peer?



TYPES OF SYSTEM ARCHITECTURES

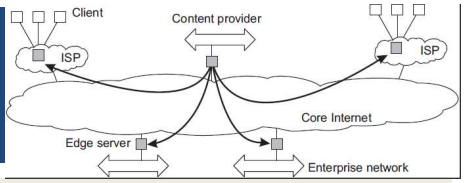
- Centralized system architectures
  - Client-server
  - Multitiered
- Decentralized peer-to-peer architectures
  - Structured
  - Unstructured
  - Hierarchically organized
- Hybrid architectures

HYBRID ARCHITECTURES



- Combine centralized server concepts with decentralized peer-to-peer models
- **Edge-server systems:**
- Ad hoc peer-to-peer devices connect to the internet through an edge server (origin server)
- Edge servers (provided by an ISP) can optimize content and application distribution by storing assets near the edge
- **Example:**
- AWS Lambda@Edge: Enables Node.js Lambda Functions to execute "at the edge" harnessing existing CloudFront Content Delivery Network (CDN) servers
- https://www.infoq.com/news/2017/07/aws-lambda-at-edge

HYBRID ARCHITECTURES - 2

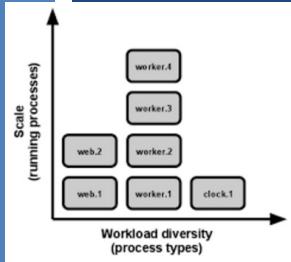


- **Fog computing:**
- Extend the scope of managed resources beyond the cloud to leverage compute and storage capacity of end-user devices
- End-user devices become part of the overall system
- Middleware extended to incorporate managing edge devices as participants in the distributed system
- Cloud → in the sky
  - *compute/resource capacity is huge, but far away…*
- Fog → (devices) on the ground
  - *compute/resource capacity is constrained and local…*

COLLABORATIVE DISTRIBUTED SYSTEM EXAMPLE

- **BitTorrent Example:**
  File sharing system – users must contribute as a file host to be eligible to download file resources
- Original implementation features hybrid architecture
- Leverages idle client network capacity in the background
- User joins the system by interacting with a central server
- Client accesses global directory from a ***tracker*** server at well known address to access torrent file
- Torrent file tracks nodes having chunks of requested file
- Client begins downloading file chunks and immediately then participates to reserve downloaded content **or network bandwidth is reduced!!**
- Chunks can be downloaded in parallel from distributed nodes

CH. 3: PROCESSES



L6.49

CHAPTER 3

- Chapter 3 titled processes
- Covers variety of distributed system implementation details
- "Grab bag" of topics

- Processes/threads
- Virtualization
- Clients
- Servers
- Code migration

THREADS

- For implementing a server (or client) threads offer many advantages vs. heavy weight processes
- What is the difference between a process and a thread?
  - Review from Operating Systems
- *Key difference*: what do threads share amongst each other that processes do not…. ?
- What are the three segments of a program stored in memory?
  - Heap segment (global memory)
  - Code segment

THREADS - 2

- Do several processes on an operating system share…
  - Heap segment?
  - Stack segment?
  - Code segment?
- Can we run multiple copies of the same code?
- These may be managed as shared pages (across processes) in memory
- Processes are isolated from each other by the OS
  - Each has a separate heap, stack, code segment

THREADS - 3

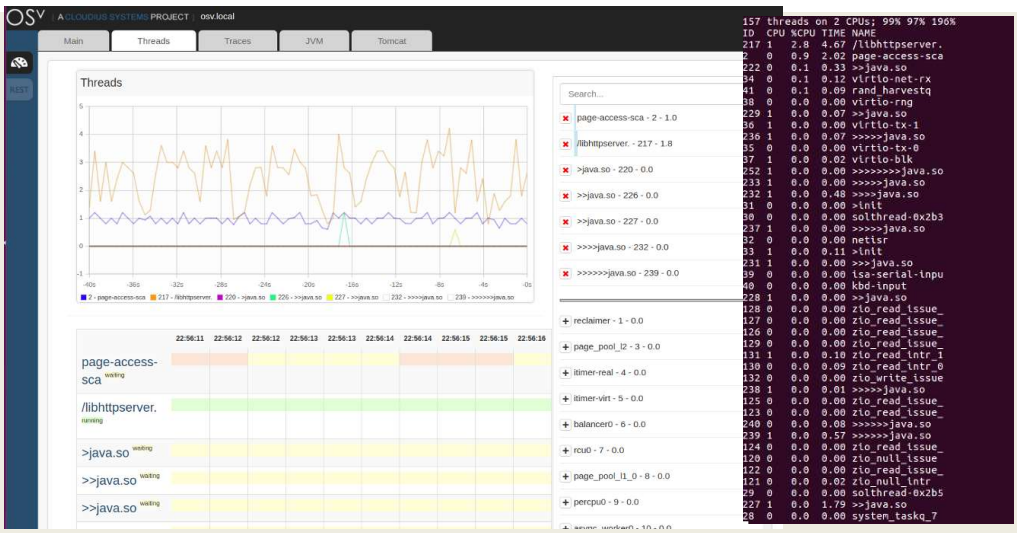
- Threads avoid the overhead of process creation
- No new heap or code segments required

- **What is a context switch?**

- Context switching among threads is considered to be more efficient than context switching processes
- Less elements to swap-in and swap-out

- Unikernels, example OSv
- Single process operating system with many threads
- Developed for the cloud to run only one application at a time

OSV: JUST THREADS



THREADS - 4

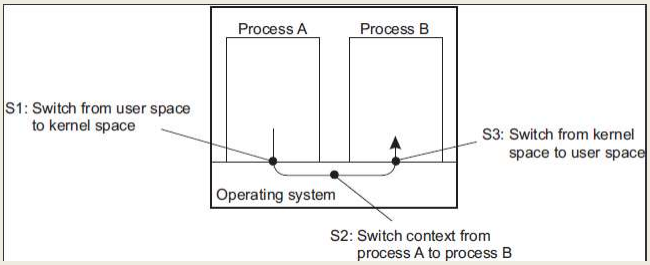
- Important implications with threads:
- (1) multi-threading should lead to performance gains
- (2) thread programming requires additional effort when threads share memory
  - Known as thread synchronization, or enabling concurrency
- Access to critical sections of code which modify shared variables must be mutually exclusive
  - No more than one thread can execute at any given time

BLOCKING THREADS

- Example: spreadsheet with formula to compute sum of column
- User modifies values in column
- Threads
1. Supports interaction (UI) activity with user
2. Updates spreadsheet calculations in parallel
3. Continually backs up spreadsheet changes to disk
- Single core CPU
  - Tasks appear as if they are performed simultaneously
- Multi core CPU
  - Tasks ***execute*** simultaneously

INTERPROCESS COMMUNICATION

- IPC – mechanism using pipes, message queues, and shared memory segments
- IPC mechanisms incur context switching
  - Process I/O must execute in kernel mode
- **For CPU context switching which is preferable?**
  (A) user space threads or (B) kernel space processes ?

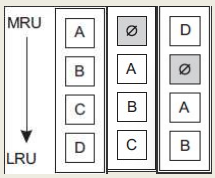


CONTEXT SWITCHING

- **Direct overhead**
  - Time spent not executing program code (user or kernel)
  - Time spent executing interrupt routines to swap memory segments of different processes (or threads) in the CPU
  - Stack, code, heap, registers, code pointers, stack pointers
  - Memory page cache invalidation
- **Indirect overhead**
  - Overhead not directly attributed to the physical actions of the context switch
  - Captures performance degradation related to the side effects of context switching
  - ***Primarily cache perturbation***

CONTEXT SWITCH – CACHE PERTURBATION

- Refers to cache reorganization that occurs as a result of context switch
- Cache is not clear, but elements from cache are removed as a result of another program running in the CPU
- 80% performance overhead from context switching results from this **"cache perturbation"**



THREADING MODELS

- **Many-to-one threading:** multiple user-level threads per process
- Thread operations (create, delete, locks) run in user mode
- Multithreaded process mapped to single schedulable entity
- Only run thread per process runs at any given time

- **What are some advantages of many-to-one threading?**

- **What are some disadvantages?**

THREADING MODELS - 2

- **One-to-one threading**: multiple kernel-level threads per process
- Thread operations (create, delete, locks) run in kernel mode
- Threads scheduled individually by the OS
- System calls required, context switches as expensive as process context switching
- Linux uses this model…

- **What are some advantages of one-to-one threading?**

- **What are some disadvantages?**

APPLICATION EXAMPLES

- Google chrome: processes
- Apache tomcat webserver: threads

- Multiprocess programming avoids synchronization of concurrent access to shared data, by providing coordination and data sharing via interprocess communication (IPC)

- Each process maintains its own private memory

- **Do distributed objects share memory?**

MULTITHREADED CLIENTS

- **Web browser**
- Uses threads to load and render portions of a web page to the user in parallel
- A client could have dozens of concurrent connections all loading in parallel

- **testFibPar.sh**
- Assignment 0 client script (GNU parallel)

- **Important benefits:**
- Several connections can be opened simultaneously
- Client: dozens of concurrent connections to the webserver all loading data in parallel

MULTIPLE THREADS

- In Linux, threads also receive a process ID (PID)
- To display threads of a process in Linux:

- Identify parent process explicitly:

- top –H –p <pid>
- htop –p <pid>
- ps –iT <pid>

- Virtualbox process ~ 44 threads
- No mapping to guest # of processes/threads

PROCESS METRICS

Disk
- dsr: disk sector reads
- dsreads: disk sector reads completed
- drm: merged adjacent disk reads
- readtime: time spent reading from disk
- dsw: disk sector writes
- dswrites: disk sector writes completed
- dwm: merged adjacent disk writes
- writetime: time spent writing to disk

CPU
- cpuUsr: CPU time in user mode
- cpuKrn: CPU time in kernel mode
- cpuIdle: CPU idle time
- cpuIoWait: CPU time waiting for I/O
- cpuIntSrvc: CPU time serving interrupts
- cpuSftIntSrvc: CPU time serving soft interrupts
- cpuNice: CPU time executing prioritized processes
- cpuSteal: CPU ticks lost to virtualized guests
- contextsw: # of context switches
- loadavg: (avg # proc / 60 secs)

Network
- nbs: network bytes sent
- nbr: network bytes received

LOAD AVERAGE

- Reported by: `top`, `htop`, `w`, `uptime`, and `/proc/loadavg`
- Updated every 5 seconds
- Average number of processes using or waiting for the CPU
- Three numbers show exponentially decaying usage for 1 minute, 5 minutes, and 15 minutes
- One minute average: exponentially decaying average
- Load average = 1 ▪ (avg last minute load) – 1/e ▪ (avg load since boot)

- 1.0 = 1-CPU core fully loaded
- 2.0 = 2-CPU cores
- 3.0 = 3-CPU cores . . .

THREAD-LEVEL PARALLELISM

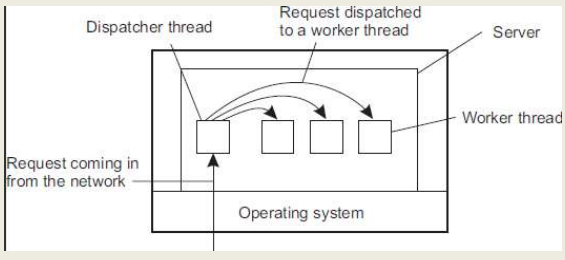
- Metric – measures degree of parallelism realized by running system, by calculating average utilization:

$$TLP = \frac{\sum_{i=1}^{N} i \cdot c_i}{1 - c_0}$$

- Ci – fraction of time that exactly I threads are executed
- N – maximum threads that can execute at any one time
- Web browsers found to have TLP from 1.5 to 2.5
- Clients for web browsing can utilize from 2 to 3 CPU cores
- Any more cores are redundant, and potentially wasteful
- **Measure TLP to understand how many CPUs to provision**

MULTITHREADED SERVERS

- Multiple threads essential for servers in distributed systems
- Even on single-core machines greatly improves performance
- Take advantage of idle/blocking time
- Two designs:
  - Generate new thread for every request
  - Thread pool – pre-initialize block of threads to service requests



SINGLE THREAD & FSM SERVERS

- Single thread server
  - A single thread handles all client requests
  - BLOCKS for I/O
  - All waiting requests are queued until thread is available
- Finite state machine
  - Server has a single thread of execution
  - I/O performing asynchronously (non-BLOCKing)
  - Server handles other requests while waiting for I/O
  - Interrupt fired with I/O completes
  - Single thread "jumps" back into context to finish request

SERVER DESIGN ALTERNATIVES

- A blocking system call implies that a thread servicing a request synchronously performs I/O
- The thread BLOCKS to wait on disk/network I/O before proceeding with request processing
- Consider the implications of these designs for responsiveness, availability, scalability. . .

Model	Characteristics
Multithreading	Parallelism, blocking I/O
Single-thread	No parallelism, blocking I/O
Finite-state machine	Parallelism, non-blocking I/O

QUESTIONS

EXTRA SLIDES