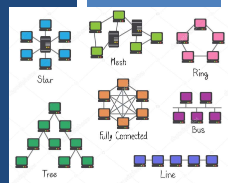
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Middleware Organization and System Architectures



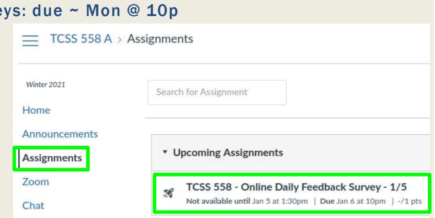
Wes J. Lloyd
School of Engineering & Technology (SET)
University of Washington - Tacoma

OBJECTIVES – 1/21

- **Questions from 1/19**
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

ONLINE DAILY FEEDBACK SURVEY

- Daily Feedback Quiz in Canvas – Available After Each Class
- Extra credit available for completing surveys ***ON TIME***
- Tuesday surveys: due by ~ Wed @ 10p
- Thursday surveys: due ~ Mon @ 10p



TCSS 558 - Online Daily Feedback Survey - 1/5

Due Jan 6 at 10pm **Points** 1 **Questions** 4
Available Jan 5 at 1:30pm - Jan 6 at 11:59pm 1 day **Time Limit** None

Question 1 0.5 pts

On a scale of 1 to 10, please classify your perspective on material covered in today's class:

1	2	3	4	5	6	7	8	9	10
Mostly Review To Me				Equal New and Review					Mostly New to Me

Question 2 0.5 pts

Please rate the pace of today's class:

1	2	3	4	5	6	7	8	9	10
Slow				Just Right					Fast

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (20 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.65** (↓ - *previous 6.74*)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.60** (↑ - *previous 5.57*)

FEEDBACK FROM 1/19

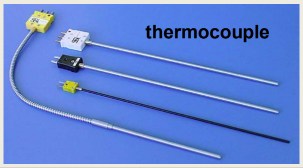
- ***In this class, there are several times that you mentioned the sensors are relatively cheaper than the micro processor. I would like to ask for a sensor network and distributed network, what are the typical prices for sensors and processors.***

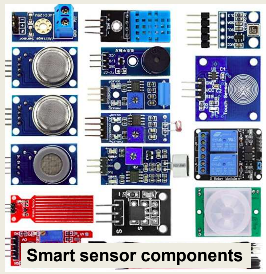
- Smart sensors are sensors with onboard compute resources capable of performing data processing before passing data on
  - Features: programmable, data aggregation

- Specifics on pricing will depend on the application and associated requirements
  - *Details go somewhat out of context for our class*

SMART SENSORS

- Building a smart sensor often involves combining an Arduino or Raspberry PI with custom sensing hardware →
- Traditional temperature sensor non-programmable ↓





FEEDBACK - 2

- ***I think the Publish-subscribe architecture remains lest clear to me. Maybe Professor could go into more details about the coordinate table.***
- ***Concepts:***
- ***Temporal:*** *is communication synchronous vs. asynchronous?*
- ***Synchronous:*** *client and server have a LIVE connection and communicate directly with each other* ***in-real-time***
  - *Think phone call & LIVE conversation*
- ***Asynchronous:*** *client and server DO NOT HAVE LIVE connection, communication is through cached messages*
  - *Think EMAIL*
- ***Referential:*** *name, as in the name of the host or IP address*
- ***Coupled:*** *communication depends on …*
- ***Decoupled:*** *communication DOES NOT depend on …*

PUBLISH-SUBSCRIBE ARCHITECTURES

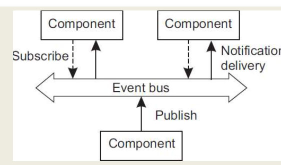
- Enables separation between processing and coordination
- Types of coordination:

	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

Publish and subscribe architectures

PUBLISH-SUBSCRIBE ARCHITECTURES - 2

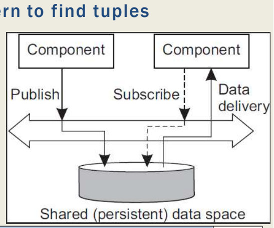
- **Event-based coordination**
- Processes do not know about each other explicitly
- **Processes:**
  - **Publish:** a notification describing an event
  - **Subscribe:** to receive notification of specific kinds of events



- Assumes subscriber is presently up (*temporally coupled*)
- Subscribers must actively **MONITOR** event bus

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

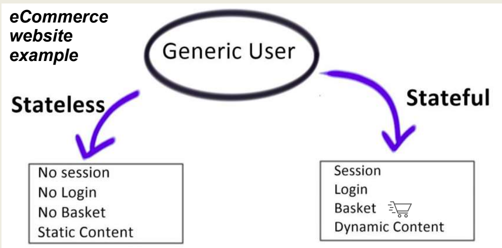


FEEDBACK

- ***Kindly suggest some additional reading material on Architectures. I would like to understand more about Temporal/Referential coupling/decoupling***
- See Chapter 6.3 on Publish-subscribe systems pg. 242-253 in **Distributed Systems: Concepts and Design**, George Coulouris, Jean Dollimore, et al. 5th Edition, Pearson, 2011.

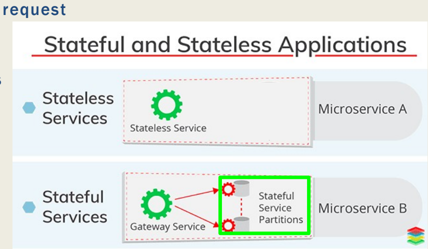
FEEDBACK - 3

- ***Can you use diagrams to explain the difference between stateless and stateful?***



FEEDBACK - 4

- Stateful webservices often have requirement to store user session information local to the server processing the request
- Gateway/load balancer needs to be aware of this



OBJECTIVES – 1/21

- Questions from 1/19
- **Assignment 0: Cloud Computing Infrastructure Tutorial**
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

ASSIGNMENT 0

- ***Preparing for Assignment 0:***
  - Establish AWS Account
    - Standard account – ***** request cloud credits from instructor *****
      - Specify "AWS CREDIT REQUEST" as subject of email
      - Include email address of AWS account
    - **AWS Educate Starter account** – some account limitations
      - https://awseducate-starter-account-services.s3.amazonaws.com/AWS_Educate_Starter_Account_Services_Supported.pdf
  - Establish local Linux/Ubuntu environment
- Task 1 – AWS account setup
- Task 2 – Working w/ Docker, creating Dockerfile for Apache Tomcat
- Task 3 – Creating a Dockerfile for haproxy
- Task 4 – Working with Docker-Machine
- Task 5 – For Submission: Testing Alternate Server Configurations

OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- **Class Activity: Architectural Styles**
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

IN-CLASS ACTIVITY: ARCHITECTURAL STYLES

CLASS ACTIVITY 2

- We will form groups of ~2-3 and enter breakout rooms
- Each group will complete a Google Doc worksheet
- Add names to Google Doc as they appear in Canvas
- Once completed, **one person** submits a PDF of the Google Doc to Canvas
- Instructor will score all group members based on the uploaded PDF file
- To get started:
  - Log into your *** **UW Google Account** ***
  - Link to shared Google Drive
  - Follow link:

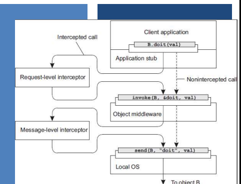
https://tinyurl.com/y43bflzs

DISTRIBUTED SYSTEM GOALS TO CONSIDER

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

OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- **Chapter 2.2: Middleware Organization**
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures



CH 2.2: MIDDLEWARE ORGANIZATION

MIDDLEWARE ORGANIZATION

- Relies on two important design patterns:
  - Wrappers
  - Interceptors
- Both help achieve the goal of openness

OBJECTIVES – 1/21

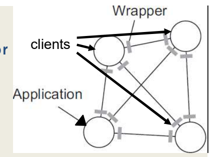
- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - **Wrappers**
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

MIDDLEWARE: WRAPPERS

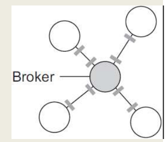
- **Wrappers (also called adapters)**
  - **WHY?:** Interfaces available from legacy software may not be sufficient for all new applications to use
  - **WHAT:** Special "frontend" components that provide interfaces for clients
  - Interface wrappers transform client requests to "implementation" (i.e. legacy software) at the component-level
  - Can then provide modern service interfaces for legacy code/systems
  - Components encapsulate (i.e. abstract) dependencies to meet all preconditions to operate and host legacy code
  - Interfaces parameterize legacy functions, abstract environment configuration (i.e. make into black box)
- Contributes towards system **OPENNESS**
- **Example: Amazon S3:** S3 HTTP REST interface
- GET/PUT/DELETE/POST: requests handed off for fulfillment

MIDDLEWARE: WRAPPERS - 2

- Inter-application communication
  - Applications may provide unique interface for every client application
- Scalability suffers
  - N applications → O(N^2) wrappers



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



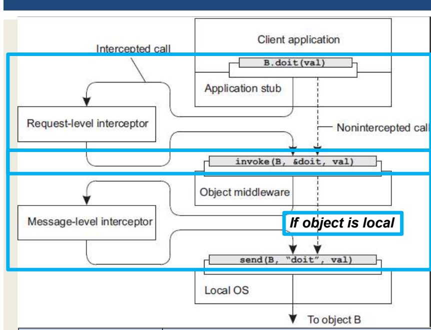
OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

MIDDLEWARE: INTERCEPTORS

- **Interceptor**
- Software construct, breaks flow of control, allows other application code to be executed
- Interceptors send calls to other servers, or to ALL servers that replicate an object while abstracting the ***distribution*** and/or ***replication***
  - Used to enable remote procedure calls (RPC), remote method invocation (RMI)
- Object A calls method belonging to object B
  - Interceptors route calls to object B regardless of location

MIDDLEWARE: INTERCEPTORS - 2



Request-level interceptor transforms: `B.doit(val)` into generic call: `invoke(B,&doit,val)`

Message-level interceptor in middleware sends message through OS (TCP/IP socket) to transfer data: `send(B,"doit",val)`

Non-intercepted:

MIDDLEWARE INTERCEPTION - METHOD

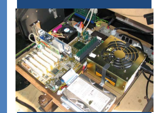
- **MIDDLEWARE**: Provides local interface matching Object B to Object A
- Object A calls Object B's method provided by local interface
- A's call is transformed into a "*generic object invocation*" by **request-level interceptor**
- "*Generic object invocation*" is transformed into a **message** by **message-level interceptor** and sent over Object A's network to Object B
- Interception automatically routes calls to all object replicas

MODIFIABLE MIDDLEWARE

- **GOAL:** 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 ?**

January 21, 2021 | TCSS558: Applied Distributed Computing [Winter 2021] School of Engineering and Technology, University of Washington - Tacoma | L6.31

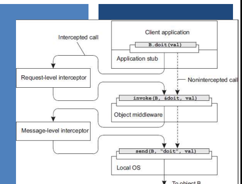
WE WILL RETURN AT 2:36PM



OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- **Chapter 2.3: System Architectures**
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

January 21, 2021 | TCSS558: Applied Distributed Computing [Winter 2021] School of Engineering and Technology, University of Washington - Tacoma | L6.33



CH 2.3: SYSTEM ARCHITECTURES

January 21, 2021 | TCSS558: Applied Distributed Computing [Winter 2021] School of Engineering and Technology, University of Washington - Tacoma | L6.34

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 an **architectural style**
- System architectures represent designs used in practice

January 21, 2021 | TCSS558: Applied Distributed Computing [Winter 2021] School of Engineering and Technology, University of Washington - Tacoma | L6.35

OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - **Centralized system architectures**
  - Decentralized peer-to-peer architectures
  - Hybrid architectures

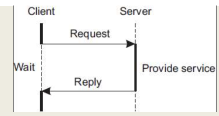
January 21, 2021 | TCSS558: Applied Distributed Computing [Winter 2021] School of Engineering and Technology, University of Washington - Tacoma | L6.36

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

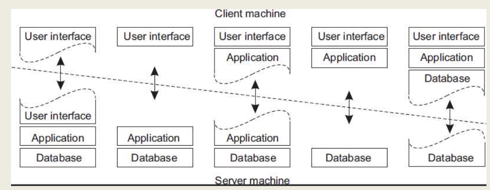
	Connectionless (UDP) *stateless*	Connection-oriented (TCP) *stateful*
Advantages		
Disadvantages		

CONNECTIONLESS VS CONNECTION ORIENTED

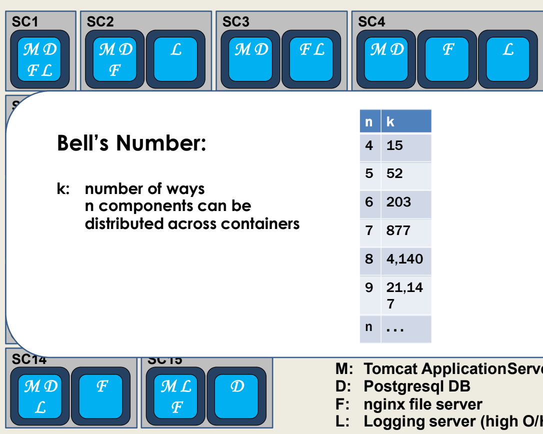
	Connectionless (UDP) *stateless*	Connection-oriented (TCP) *stateful*
Advantages	• Fast to communicate (no connection overhead) • Broadcast to an audience • Network bandwidth savings	• Message delivery confirmation • Idempotence not required • Messages automatically resent - if client (or network) is 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?



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	…

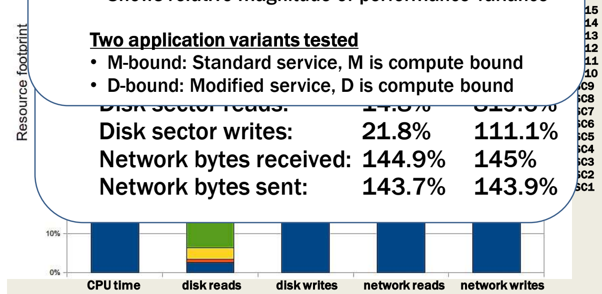
M: Tomcat ApplicationServer
D: Postgresql DB
F: nginx file server
L: Logging server (high O/H)

Resource 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

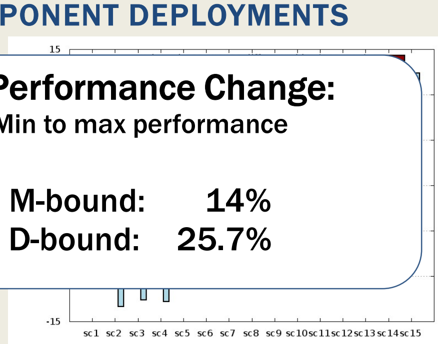
Disk sector writes:	21.8%	111.1%
Network bytes received:	144.9%	145%
Network bytes sent:	143.7%	143.9%



PERFORMANCE IMPLICATIONS OF COMPONENT DEPLOYMENTS

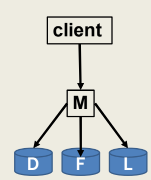
∆ Performance Change:
Min to max performance

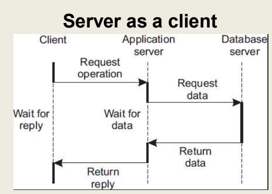
M-bound: 14%
D-bound: 25.7%



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

MULTITIERED RESOURCE SCALING - 2

- **Horizontal distribution cont’d**
  - Sharding: portions of a database map” to a specific server
  - Distributed hash table
  - Or replica servers

OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - **Decentralized peer-to-peer architectures**
  - Hybrid architectures

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 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)
```

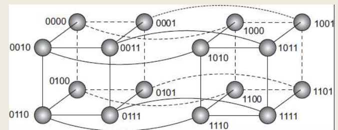
- Hash function "generates" a unique key based on the data
- No two data elements will have the same key (hash)
- System supports data lookup via key
- ***Any*** node can receive and resolve the request
- Lookup function determines which node stores the key

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

- Node forwards request to node with the data

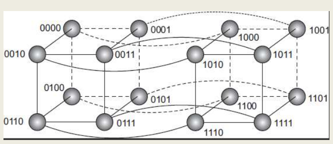
FIXED HYPERCUBE EXAMPLE

- 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?**

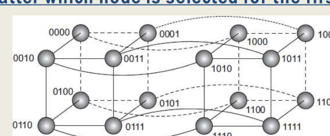


WHICH CONNECTOR LEADS TO THE SHORTEST PATH?

- **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)

- **Does it matter which node is selected for the first hop?**

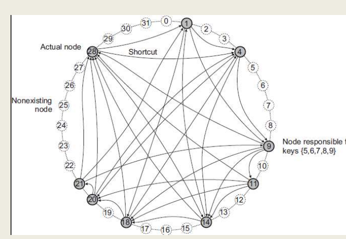


DYNAMIC TOPOLOGY

- Fixed hypercube requires static topology
  - Nodes cannot join or leave
- Relies on symmetry of number of nodes
- Can force the DHT to a certain size
- Chord system – DHT (again 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 adhoc list of neighbors
- Facilitates nodes frequently joining, leaving, adhoc 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

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 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 start "n" random walks simultaneously to reduce search time
  - As few as n=16..64 random walks sufficient to reduce search time (LV et al. 2002)
  - Timeout required - need to coordinate stopping network-wide walk when data is found…

SEARCHING FOR DATA - 3

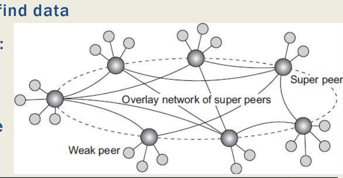
- **Policy-based search methods**
- Incorporate history and knowledge about the adhoc network ***at the node-level*** to enhance effectiveness of queries
- Nodes maintain lists of preferred neighbors which often succeed at resolving queries
- Favor neighbors having highest number of neighbors
  - Can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

HIERARCHICAL PEER-TO-PEER NETWORKS

- **Problem:**
  Adhoc 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

HIERARCHICAL 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?



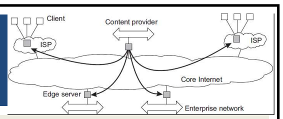
OBJECTIVES – 1/21

- Questions from 1/19
- Assignment 0: Cloud Computing Infrastructure Tutorial
- Class Activity: Architectural Styles
- Chapter 2.2: Middleware Organization
  - Wrappers
  - Interceptors
- Chapter 2.3: System Architectures
  - Centralized system architectures
  - Decentralized peer-to-peer architectures
  - **Hybrid architectures**

TYPES OF SYSTEM ARCHITECTURES

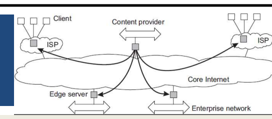
- Centralized system architectures
  - Client-server
  - Multitiered
- Decentralized peer-to-peer architectures
  - Structured
  - Unstructured
  - Hierarchically organized
- Hybrid architectures

HYBRID ARCHITECTURES



- Combine centralized server concepts with decentralized peer-to-peer models
- **Edge-server systems:**
- Adhoc 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

QUESTIONS