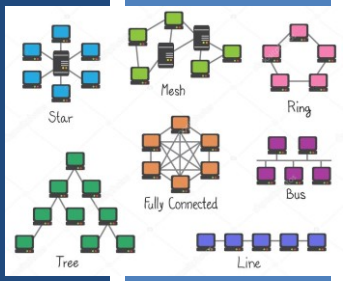
# TCSS 558: APPLIED DISTRIBUTED COMPUTING

## Ch. 4 - Communication

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




1

## OBJECTIVES – 2/15

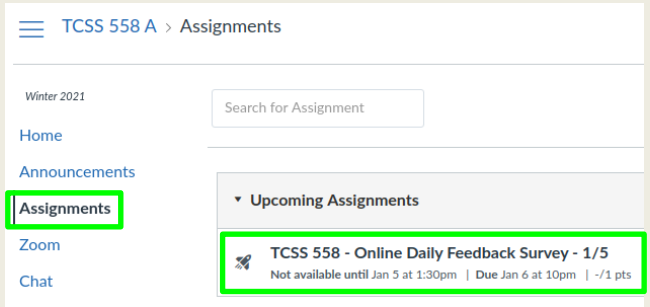
- Questions from 2/13
- Midterm Review
- Assignment 2: Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-review)
  - Chapter 4.3: Message Oriented Communication



2

## 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






3

## 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 |



4

## MATERIAL / PACE

- Please classify your perspective on material covered in today's class (19 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.16** (↑ - *previous 5.45*)

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 6.11** (↑ - *previous 5.29*)



5

## QUESTIONS FROM 2/13



6

## CSS TENURE TRACK FACULTY CANDIDATE RESEARCH SEMINARS – EXTRA CREDIT

- **Week 8:**
  - Tuesday February 20 – 12:30pm –room TBA
  - Thursday February 22 – 12:30pm –room TBA
  - Friday February 23 – 1:30pm –room TBA
- **Week 9 (every day):**
  - Feb 26, 27, 28, 29, Mar 1 – 1:30pm –room TBA
- **Week 10 (Monday and Tuesday):**
  - Mar 4, 5 – 1:30pm – room TBA

**In Winter 23, the final exam scores ran approx. 10 pts on average below the W'24 midterm scores**

- Earn up to 30 buffer points added to the Final Exam score
- Earn 3 points for each seminar attended
- Buffer points replace missed points on the Final Exam
- Once the Final Exam score = 100%, additional points do not push the Final Exam score above 100%
- Buffer points will not impact the course curve for the Final Exam
- Any course curve will be applied before buffer points



7

## OBJECTIVES – 2/15

- Questions from 2/13
- **Midterm Review**
- Assignment 2: Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-review)
  - Chapter 4.3: Message Oriented Communication



8

## MIDTERM

- Review of questions



12

## OBJECTIVES – 2/15

- Questions from 2/13
- Midterm Review
- **Assignment 2: Key Value Store**
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-review)
  - Chapter 4.3: Message Oriented Communication

**This seems fairly normally distributed.**



13

## ASSIGNMENT 2

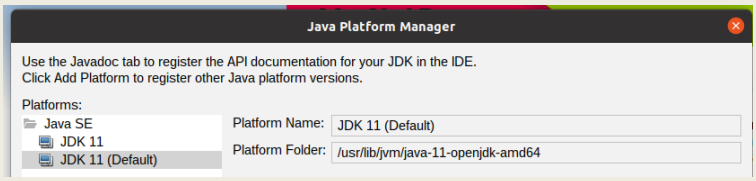
- **Find Teammates**: signup posted on Canvas under 'People'
- GenericNode.tar.gz includes Dockerfile examples
- GenericNode.tar.gz assumes Java 11
- TCP/UDP/RMI Key Value Store
- Implement a "GenericNode" project which assumes the role of a client or server for a Key/Value Store
- Recommended in Java 11 LTS
- Client node program interacts with server node to put, get, delete, or list items in a key/value store



14

## USING JAVA 11 IN NETBEANS

- In Netbeans IDE, under Tools menu, 'Java Platforms', be sure to install and select JDK 11




- On left-hand Project menu, right-click on '**GenericNode**' project
- Select Properties
- Under Build | Compile, be sure Java Platform is JDK 11
- Under Sources, be sure Source/Binary Format is 11



15

## OBJECTIVES – 2/15

- Questions from 2/13
- Midterm Review
- Assignment 2: Key Value Store
- **Chapter 4: Communication**
  - **Chapter 4.1: Foundations**
  - Chapter 4.2: RPC (light-review)
  - Chapter 4.3: Message Oriented Communication



16


17

# CHAPTER 4

- 4.1 Foundations
  - Protocols
  - Types of communication
- 4.2 Remote procedure call
- 4.3 Message-oriented communication
  - Socket communication
  - Messaging libraries
  - Message-Passing Interface (MPI)
  - Message-queueing systems
  - Examples
- 4.4 Multicast communication
  - Flooding-based multicasting
  - Gossip-based data dissemination

***Reviews and builds on content from Ch. 2/3***



18

# CH. 4.1: FOUNDATIONS



19

## TYPES OF COMMUNICATION

- Persistent communication
  - Message submitted for transmission is stored by communication middleware as long as it takes to deliver it
  - Example: email system (SMTP)
  - Receiver can be offline when message sent
  - Temporal decoupling (delayed message delivery)
- Transient communication
  - Message stored by middleware only as long as sender/receiver applications are running
  - If recipient is not active, message is dropped
  - Transport level protocols typically are transient (*no msg storage*)
- **What OSI protocol level is the SMTP Protocol?**



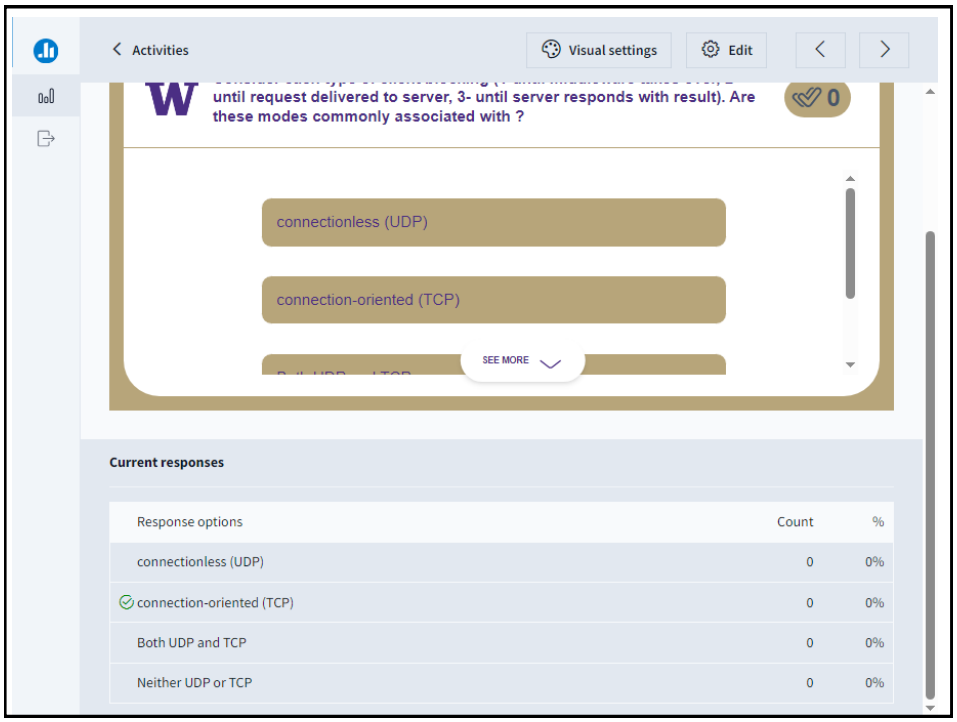
20

## TYPES OF COMMUNICATION - 2

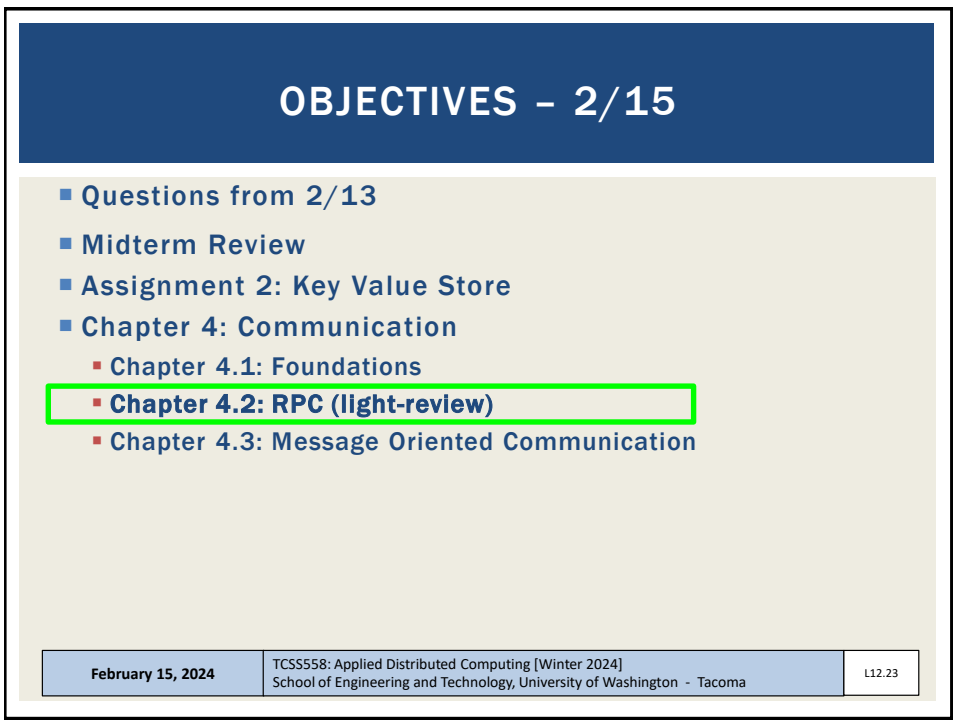
- Asynchronous communication
  - Client does not block, continues doing other work
- Synchronous communication
  - Client blocks and waits
- Three types of blocking (*synchronous*)
  1. SHORTEST:
     Until middleware notifies it will take over delivering ***request***
  2. MEDIUM:
     Sender may block until ***request*** has been delivered
  3. LONGEST:
     Sender waits until ***request*** is processed and result is returned
- Persistence + synchronization (blocking)
  - Common scheme for message-queueing systems
  - ***Publish message to queue***: block until message delivered to queue



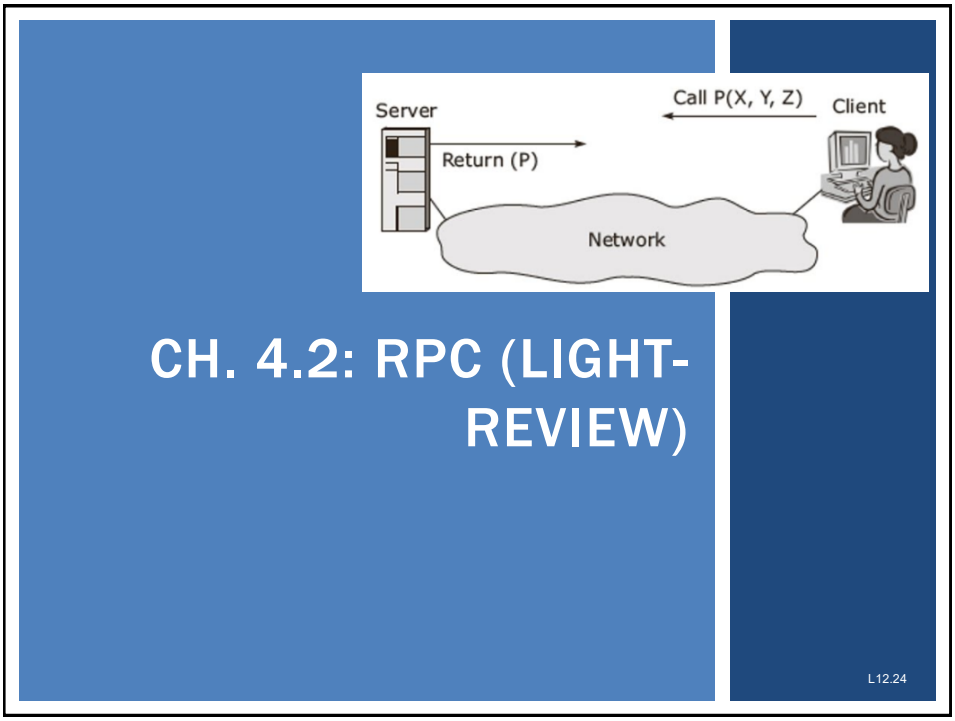
21


Activities
Visual settings
Edit
until request delivered to server, 3- until server responds with result). Are these modes commonly associated with ?
0
connectionless (UDP)
connection-oriented (TCP)
SEE MORE
Current responses
Response options
Count
%
connectionless (UDP)
0
0%
connection-oriented (TCP)
0
0%
Both UDP and TCP
0
0%
Neither UDP or TCP
0
0%

22


OBJECTIVES – 2/15
Questions from 2/13
Midterm Review
Assignment 2: Key Value Store
Chapter 4: Communication
Chapter 4.1: Foundations
Chapter 4.2: RPC (light-review)
Chapter 4.3: Message Oriented Communication
February 15, 2024
TCSS558: Applied Distributed Computing [Winter 2024]
School of Engineering and Technology, University of Washington - Tacoma
L12.23

23




24

## RPC – REMOTE PROCEDURE CALL

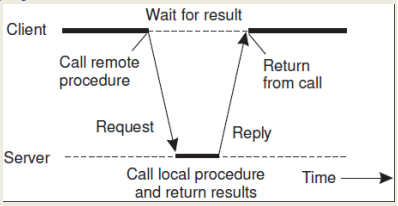
- In a nutshell,
- Allow programs to call procedures on other machines
- Process on **machine A** calls procedure on **machine B**
- Calling process on **machine A** is suspended
- Execution of the called procedure takes place on **machine B**
- Data transported from caller **(A)** to provider **(B)** and back **(A)**.
- No message passing is visible to the programmer
- **Distribution transparency**: make remote procedure call look like a local one
- `newlist = append(data, dbList)`



25

## RPC - 2

- Transparency enabled with client and server "stubs"
- Client has "stub" implementation of the server-side function
- Interface exactly same as server side
- But client **DOES NOT HAVE THE IMPLEMENTATION**
- **Client stub**: packs parameters into message, sends ***request*** to server. Call blocks and waits for reply
- **Server stub**: transforms incoming ***request*** into local procedure call
- Blocks to wait for ***reply***
- Server stub unpacks ***request***, calls server procedure
- ***It's as if the routine were called locally***






26

## RPC - 3

- Server packs procedure ***results*** and sends back to client.
- Client "***request***" call unblocks and data is unpacked
- Client can't tell method was called remotely over the network… ***except for network latency…***
- Call abstraction enables clients to invoke functions in alternate languages, on different machines
- Differences are handled by the RPC "framework"



27

## RPC STEPS

1. Client procedure calls client stub
2. Client stub builds message and calls OS
3. Client's OS send message to remote OS
4. Server OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server performs work, returns results to server-side stub
7. Server stub packs results in messages, calls server OS
8. Server OS sends message to client's OS
9. Client's OS delivers message to client stub
10. Client stub unpacks result, returns to client

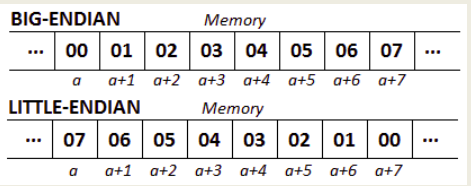


## PARAMETER PASSING

- **STUBS**: take parameters, pack into a message, send across network
- Parameter marshaling:
- `newlist = append(data, dbList)`
- Two parameters must be sent over network and correctly interpreted
- Message is transferred as a series of bytes
- Data is serialized into a "stream" of bytes
- Must understand how to unmarshal (unserialize) data
- Processor architectures vary with how bytes are numbered: Intel (right→left), older ARM (left→right)

## RPC: BYTE ORDERING

- Big-Endian: write bytes left to right (ARM processors)
- Little-endian: write bytes right to left (x86 processors)
- Networks: typically transfer data in Big-Endian form
- Solution: transform data to machine/network independent format
- Marshaling/unmarshaling: transform data to neutral format






30

## RPC: PASS-BY-REFERENCE

- Passing by value is straightforward
- Passing by reference is challenging
- Pointers only make sense on local machine owning the data
- Memory space of client and server are different

- Solutions to **RPC pass-by-reference**:

1. Forbid pointers altogether
2. Replace pass-by-reference with pass-by-value
   - Requires transferring entire object/array data over network
   - **Read-only optimization**: don't return data if unchanged on server
3. Passing global references
   - Example: file handle to file accessible by client and server via shared file system



31

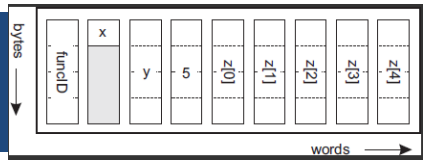
## RPC: DEVELOPMENT SUPPORT

- Let developer specify which routines will be called remotely
  - Automate client/server side stub generation for these routines
- Embed remote procedure call mechanism into the programming language
  - E.g. Java RMI



32

## STUB GENERATION




- `void func(char x; float y; int z[5])`
- 1-byte character transmits with 3-padded bytes
- Float sent as whole word (4-bytes)
  - Array as group of words, proceed by word describing length
  - Client stub must package data in specific format
  - Server stub must receive and unpackage in specific format
- Client and server must agree on representation of simple data structures: int, char, floats w/ little endian
- RPC clients/servers: must agree on protocol
  - TCP? UDP?



33

## STUB GENERATION - 2

- Interfaces are specified using an Interface Definition Language (IDL)
- Interface specifications in IDL are used to generate language specific stubs
- IDL is compiled into client and server-side stubs
- Much of the plumbing for RPC involves maintaining boilerplate-code



34

## LANGUAGE BASED SUPPORT

- Leads to simpler application development
- Helps with providing access transparency
  - Differences in data representation, and how object is accessed
  - Inter-language parameter passing issues resolved:
    → ***just 1 language***
- Well known example: ***Java Remote Method Invocation*** RPC equivalent embedded in Java

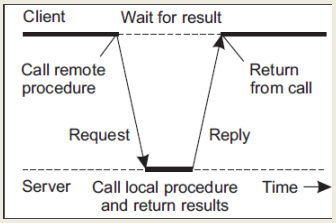


35

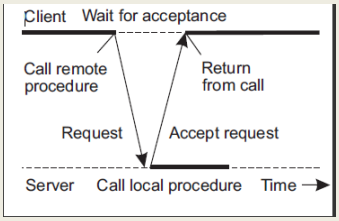
# RPC VARIATIONS

- RPC: client typically blocks until reply is returned
- Strict blocking ***unnecessary*** when there is no result
- **Asynchronous RPCs**
  - When no result, server can immediately send reply

**Client/server synchronous RPC**




**Client/server asynchronous RPC**






36

# RPC VARIATIONS – 2

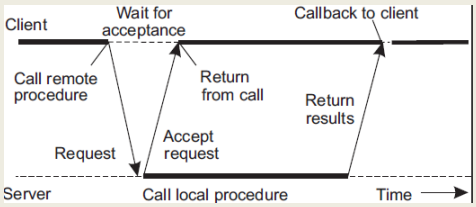
- What are tradeoffs for synchronous vs. asynchronous procedure calls?
  - For a local program
  - For a distributed program (system)
- Use cases for asynchronous procedure calls
  - Long running jobs allow client to perform alternate work in background (in parallel)
  - Client may need to make multiple service calls to multiple server backends at the same time…



37

## TYPES OF ASYNCHRONOUS RPC

- **Deferred synchronous RPC**
  - Server performs ***CALLBACK*** to client
  - Client, upon making call, spawns separate thread which blocks and waits for call



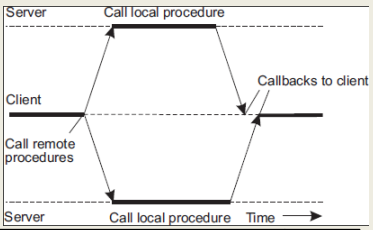

- **One-way RPCs**
  - Client **does not wait** for ***any*** server acknowledgement – it just goes…
- **Client polling**
  - Client (*using separate thread*) continually polls server for result



38

## MULTICAST RPC

- Send RPC request *simultaneously* to group of servers
- Hide that multiple servers are involved
- Consideration:
  ***Does the client need all results or just one?***
- Use cases:
  - Fault tolerance – wait for just one
  - Replicate execution – verify results, *use first result*
  - Divide and conquer - multiple RPC calls work in parallel on different parts of dataset, client aggregates results






39

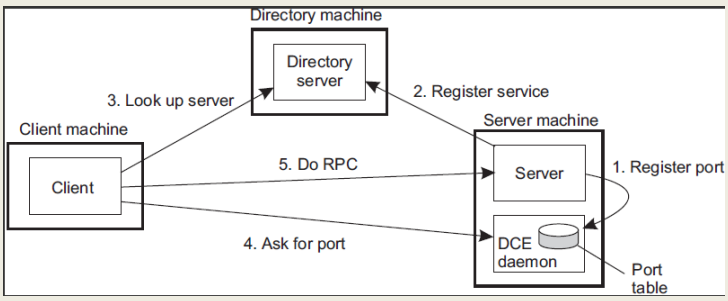
## RPC EXAMPLE: DISTRIBUTED COMPUTING ENVIRONMENT (DCE)

- DCE: basis for Microsoft's distributed computing object model (DCOM)
- Used in Samba, ***cross-platform*** file and print sharing via RPC
- Middleware system – provides layer of abstraction between OS and distributed applications
- Designed for Unix, ported to ***all*** major operating systems
- Install DCE middleware on set of heterogeneous machines – distributed applications can then access shared resources to:
  - Mount a windows file system on Linux
  - Share a printer connected to a Windows server
- Uses client/server model
- All communication via RPC
- DCE daemon tracks participating machines, ports



40

## DCE CLIENT-TO-SERVER BINDING




- Server name comes from directory server
- Server port comes from DCE daemon
  - DCE daemon has a well known port # client already knows



41

## EXTRA: DCE – CLIENT/SERVER DEVELOPMENT

1. Create Interface definition language (IDL) files
   - IDL files contain Globally unique identifier (GUID)
   - GUIDs must match: client and server compare GUIDs to verify proper versions of the distributed object
   - 128-bit binary number
2. Next, add names of remote procs and params to IDL
3. Then compile the IDL files
   ***Compiler generates:***
   - Header file (interface.h in C)
   - Client stub
   - Server stub



42

## EXTRA: DCE – BINDING CLIENT TO SERVER

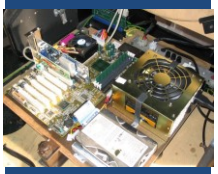
- For a client to call a server, server must be registered
  - *Java: uses RMI registry*
- Client process to search for RMI server:
  1. Locate the server's host machine
  2. Locate the server (i.e. process) on the host
- Client must discover the server's RPC port
- **DCE daemon:** maintains table of (server,port) pairs
- When servers boot:

1. Server asks OS for a port, registers port with DCE daemon
2. Also, server registers with directory server, separate server that tracks DCE servers



43

# WE WILL RETURN AT 2:58 PM



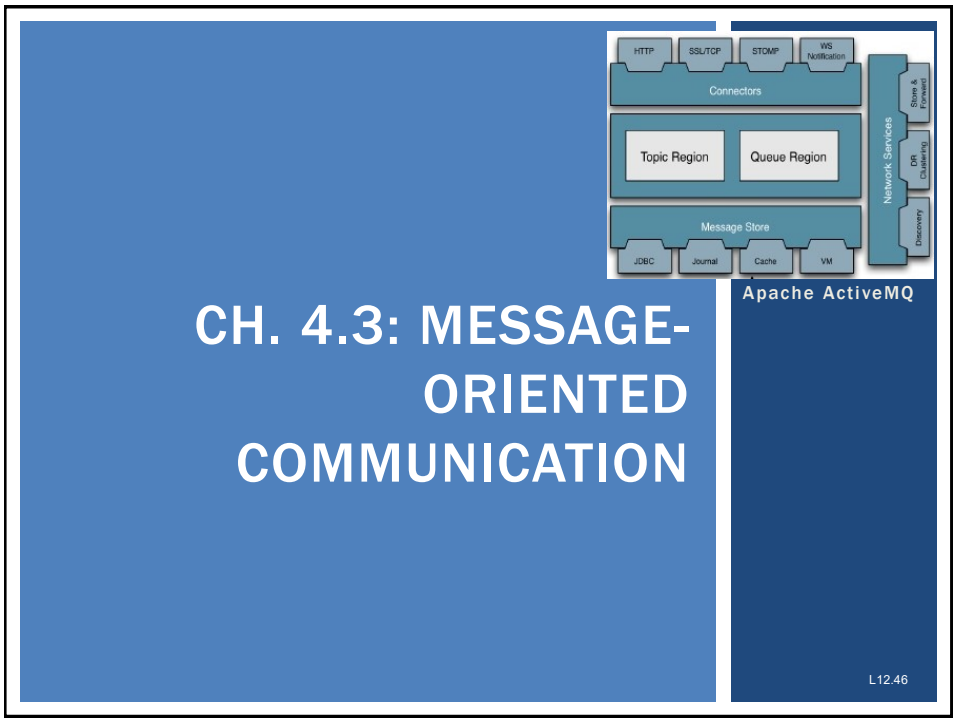
44

## OBJECTIVES – 2/15

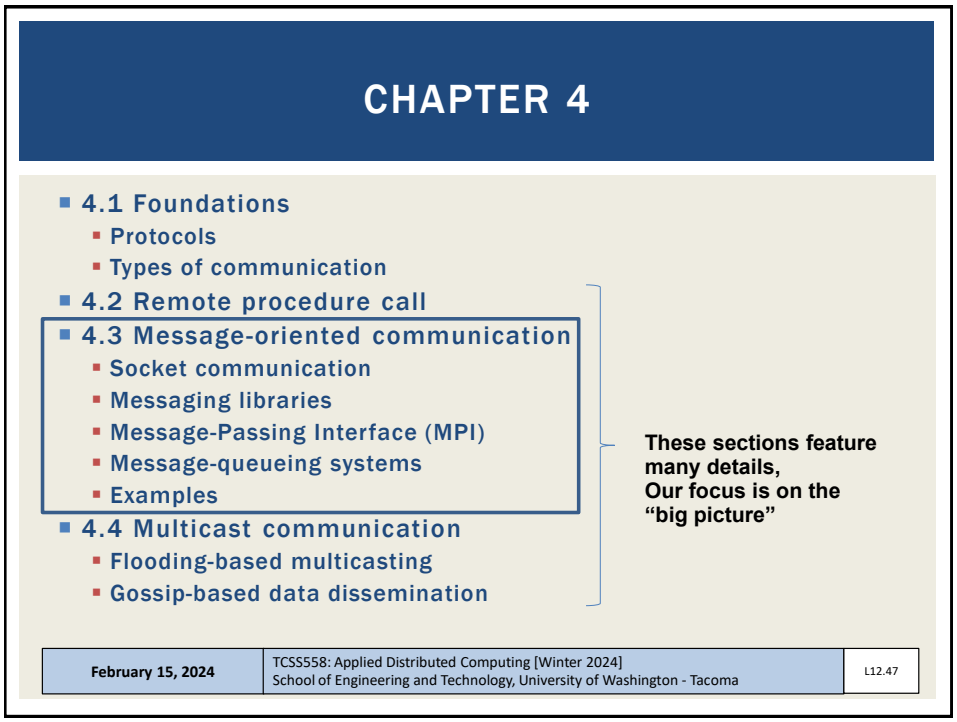
- Questions from 2/13
- Midterm Review
- Assignment 2: Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-review)
  - **Chapter 4.3: Message Oriented Communication**



45


HTTP
SSL/TCP
STOMP
WS Notification
Connectors
Topic Region
Queue Region
Message Store
JDBC
Journal
Cache
VM
Network Services
Store & Forward
DR Clustering
Discovery
Apache ActiveMQ
CH. 4.3: MESSAGE-ORIENTED COMMUNICATION
L12.46

46


CHAPTER 4
4.1 Foundations
Protocols
Types of communication
4.2 Remote procedure call
4.3 Message-oriented communication
Socket communication
Messaging libraries
Message-Passing Interface (MPI)
Message-queueing systems
Examples
4.4 Multicast communication
Flooding-based multicasting
Gossip-based data dissemination
These sections feature many details, Our focus is on the "big picture"
February 15, 2024
TCSS558: Applied Distributed Computing [Winter 2024]
School of Engineering and Technology, University of Washington - Tacoma
L12.47

47

## MESSAGE ORIENTED COMMUNICATION

- RPC assumes that the *client* and *server* are running ***at the same time…*** *(temporally coupled)*
- RPC communication is typically ***synchronous***

- When client and server are not running at the same time
- Or when communications should not be **blocked**…

- **This is a use case for message-oriented communication**
  - Synchronous vs. asynchronous
  - Messaging systems
  - Message-queueing systems



48

## SOCKETS

- Communication end point
- Applications can read / write data to
- Analogous to file streams for I/O, but ***network streams***

| Operation | Description |
|---|---|
| socket | Create a new communication end point |
| bind | Attach local address to socket (IP / port) |
| listen | Tell OS what max # of pending connection requests should be |
| accept | Block caller until a connection request arrives |
| connect | Actively attempt to establish a connection |
| send | Send some data over the connection |
| receive | Receive some data over the connection |
| close | Release the connection |



49

## SOCKETS - 2

- Servers execute 1st - 4 operations (socket, bind, listen, accept)
- Methods refer to C API functions
- Mappings across different libraries will vary (*e.g. Java*)

| Operation | Description |
|---|---|
| socket | Create a new communication end point |
| bind | Attach local address to socket (IP / port) |
| listen | Tell OS what max # of pending connection requests should be |
| accept | Block caller until a connection request arrives |
| connect | Actively attempt to establish a connection |
| send | Send some data over the connection |
| receive | Receive some data over the connection |
| close | Release the connection |



50

## SERVER SOCKET OPERATIONS

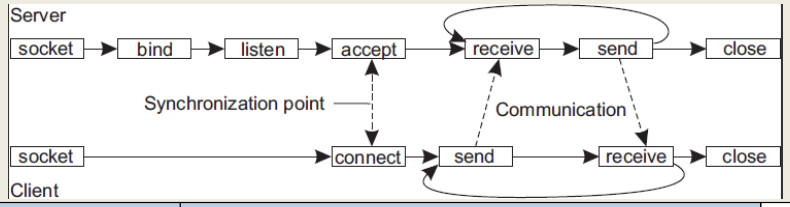
- **Socket**: creates new communication end point
- **Bind**: associated IP and port with end point
- **Listen**: for connection-oriented communication, non-blocking call reserves buffers for specified number of pending connection requests server is willing to accept
- **Accept**: blocks until connection request arrives
  - Upon arrival, new socket is created matching original
  - Server spawns thread, or forks process to service incoming request
  - Server continues to wait for new connections on original socket



51

## CLIENT SOCKET OPERATIONS

- **Socket**: Creates socket client uses for communication
- **Connect**: Server transport-level address provided, client blocks until connection established
- **Send**: Supports sending data (to: server/client)
- **Receive**: Supports receiving data (from: server/client)
- **Close**: Closes communication channel
  - Analogous to closing a file stream






52

## SOCKET COMMUNICATION

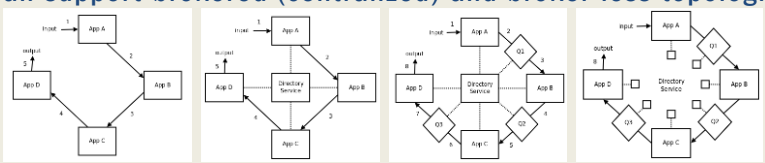
- Sockets provide primitives for implementing your own TCP/UDP communication protocols
- Directly using sockets for transient (non-persisted) messaging is very basic, can be brittle
  - Easy to make mistakes…
- Any extra communication facilities must be implemented by the application developer
- More advanced approaches are desirable
  - E.g. frameworks with support common desirable functionality



53

## ZEROMQ – SOCKET LIBRARY

- (0MQ) High performance intelligent **socket library**
- ***zero broker, zero latency, zero admin, zero cost, zero waste***
- Provides a message queue
- ***Builds upon*** functionality of traditional sockets
- Implementation in C++
  - 30+ language bindings provided
- Enables support for various messaging patterns
- Can support brokered (centralized) and broker-less topologies



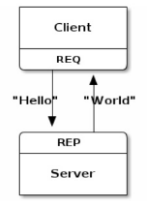


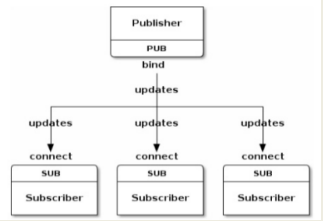
## ZEROMQ – 2

- ZeroMQ is **TCP-connection-oriented communication**
- Provides socket-like primitives with more functionality
  - Basic socket operations ***abstracted*** away
  - Supports many-to-one, one-to-one, and one-to-many connections
  - ***Multicast*** connections (one-to-many – single server socket simultaneously "connects" to multiple clients)
- Asynchronous messaging
- Supports pairing sockets to support communication patterns

# ZEROMQ - PATTERNS

- **Request-reply pattern**
  - Traditional client-server communication (e.g. RPC)
  - Client: request socket (**REQ**)
  - Server: reply socket (**REP**)
- **Publish-subscribe pattern**
  - Clients **subscribe** to messages **published** by servers
  - As in event-based coordination (Ch. 1)
  - Supports multicasting messages from server to multiple
  - Client: subscribe socket (**SUB**)
  - Server: publish socket (**PUB**)



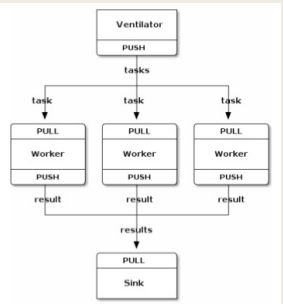






56

# ZEROMQ – PATTERNS - 2

- **Pipeline pattern (FIFO-queue)**
  - Analogous to a producer/consumer bounded buffer
  - Producing processes generate results, push to pipe
  - Consuming processes consume results, pull from pipe
  - Producers: push socket (**PUSH** socket)
  - Consumers: pull socket (**PULL** socket)
  - Push- distributes messages to all pull clients evenly
  - Consumers pull results from pipe and push results downstream






57

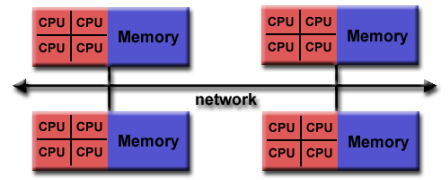
## QUEUEING ALTERNATIVES

- Cloud services
  - Amazon Simple Queueing Service (SQS)
  - Azure service bus
- Open source frameworks
  - Nanomsg
  - ZeroMQ



## MESSAGE PASSING INTERFACE (MPI)

- MPI introduced – version 1.0 March 1994
- Message passing API for parallel programming: ***supercomputers***
- Communication protocol for parallel programming for: Supercomputers, High Performance Computing (HPC) clusters
- Point-to-point and collective communication
- Goals: high performance, scalability, portability
- Most implementations in C, C++, Fortran



## MOTIVATIONS FOR MPI

- Motivation: sockets insufficient for interprocess communication on large scale HPC compute clusters and super computers
  - Sockets at the wrong level of abstraction
  - Sockets designed to communicate over the network using general purpose TCP/IP stacks
  - Not designed for proprietary protocols
  - Not designed for high-speed interconnection networks used by supercomputers, HPC-clusters, etc.
  - Better buffering and synchronization needed



60

## MOTIVATIONS FOR MPI - 2

- Supercomputers had proprietary communication libraries
  - Offer a wealth of efficient communication operations
- All libraries mutually incompatible
- Led to significant portability problems developing parallel code that could migrate across supercomputers
- Led to development of MPI
  - To support transient (non-persistent) communication for parallel programming



61

# MPI FUNCTIONS / DATATYPES

- Very large library, v1.0 (1994) 128 functions
- Version 3 (2015) 440+
- MPI data types:
- Provide common mappings

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

| | | | |
|---|---|---|---|
| MPI_ABORT | MPI_ADDRESS | MPI_ALLGATHER | MPI_ALLGATHERV |
| MPI_ALLREDUCE | MPI_ALLTOALL | MPI_ALLTOALLV | MPI_ATTR_DELETE |
| MPI_ATTR_GET | MPI_ATTR_PUT | MPI_BARRIER | MPI_BCAST |
| MPI_BSEND | MPI_BSEND_INIT | MPI_BUFFER_ATTACH | MPI_BUFFER_DETACH |
| MPI_CANCEL | MPI_CARTDIM_GET | MPI_CART_COORDS | MPI_CART_CREATE |
| MPI_CART_GET | MPI_CART_MAP | MPI_CART_RANK | MPI_CART_SHIFT |
| MPI_CART_SUB | MPI_COMM_COMPARE | MPI_COMM_CREATE | MPI_COMM_DUP |
| MPI_COMM_FREE | MPI_COMM_GROUP | MPI_COMM_RANK | MPI_COMM_REMOTE_GROUP |
| MPI_COMM_REMOTE_SIZE | MPI_COMM_SIZE | MPI_COMM_SPLIT | MPI_COMM_TEST_INTER |
| MPI_DIMS_CREATE | MPI_ERRHANDLER_CREATE | MPI_ERRHANDLER_FREE | MPI_ERRHANDLER_GET |
| MPI_ERRHANDLER_SET | MPI_ERROR_CLASS | MPI_ERROR_STRING | MPI_FINALIZE |
| MPI_GATHER | MPI_GATHERV | MPI_GET_COUNT | MPI_GET_ELEMENTS |
| MPI_GET_PROCESSOR_NAME | MPI_GRAPHDIMS_GET | MPI_GRAPH_CREATE | MPI_GRAPH_GET |
| MPI_GRAPH_MAP | MPI_GRAPH_NEIGHBORS | MPI_GRAPH_NEIGHBORS_COUNT | MPI_GROUP_COMPARE |
| MPI_GROUP_DIFFERENCE | MPI_GROUP_EXCL | MPI_GROUP_FREE | MPI_GROUP_INCL |
| MPI_GROUP_INTERSECTION | MPI_GROUP_RANGE_EXCL | MPI_GROUP_RANGE_INCL | MPI_GROUP_RANK |
| MPI_GROUP_SIZE | MPI_GROUP_TRANSLATE_RANKS | MPI_GROUP_UNION | MPI_IBSEND |
| MPI_INIT | MPI_INITIALIZED | MPI_INTERCOMM_CREATE | MPI_INTERCOMM_MERGE |
| MPI_IPROBE | MPI_IRECV | MPI_IRSEND | MPI_ISEND |
| MPI_ISSEND | MPI_KEYVAL_CREATE | MPI_KEYVAL_FREE | MPI_OP_CREATE |
| MPI_OP_FREE | MPI_PACK | MPI_PACK_SIZE | MPI_PCONTROL |
| MPI_PROBE | MPI_RECV | MPI_RECV_INIT | MPI_REDUCE |
| MPI_REDUCE_SCATTER | MPI_REQUEST_FREE | MPI_RSEND | MPI_RSEND_INIT |
| MPI_SCAN | MPI_SCATTER | MPI_SCATTERV | MPI_SEND |
| MPI_SENDRECV | MPI_SENDRECV_REPLACE | MPI_SEND_INIT | MPI_SSEND |
| MPI_SSEND_INIT | MPI_START | MPI_STARTALL | MPI_TEST |
| MPI_TESTALL | MPI_TESTANY | MPI_TESTSOME | MPI_TEST_CANCELLED |
| MPI_TOPO_TEST | MPI_TYPE_COMMIT | MPI_TYPE_CONTIGUOUS | MPI_TYPE_EXTENT |
| MPI_TYPE_FREE | MPI_TYPE_HINDEXED | MPI_TYPE_HVECTOR | MPI_TYPE_INDEXED |
| MPI_TYPE_LB | MPI_TYPE_SIZE | MPI_TYPE_STRUCT | MPI_TYPE_UB |
| MPI_TYPE_VECTOR | MPI_UNPACK | MPI_WAIT | MPI_WAITALL |
| MPI_WAITANY | MPI_WAITSOME | MPI_WTICK | MPI_WTIME |



62

# COMMON MPI FUNCTIONS

- MPI - no recovery for process crashes, network partitions
- Communication among grouped processes: `(groupID, processID)`
- IDs used to route messages in place of IP addresses

| Operation | Description |
|---|---|
| MPI_bsend | Append outgoing message to a local send buffer |
| MPI_send | Send message, wait until copied to local/remote buffer |
| MPI_ssend | Send message, wat until transmission starts |
| MPI_sendrecv | Send message, wait for reply |
| MPI_isend | Pass reference to outgoing message and continue |
| MPI_issend | Pass reference to outgoing messages, wait until receipt start |
| MPI_recv | Receive a message, block if there is none |
| MPI_irecv | Check for incoming message, **do not block!** |



63

## MESSAGE-ORIENTED-MIDDLEWARE

- **Message-queueing systems**
  - Provide extensive support for ***persistent*** asynchronous communication
  - In contrast to transient systems
  - Temporally decoupled: messages are eventually delivered to recipient queues
- Message transfers may take minutes vs. sec or ms
- Each application has its own private queue to which other applications can send messages

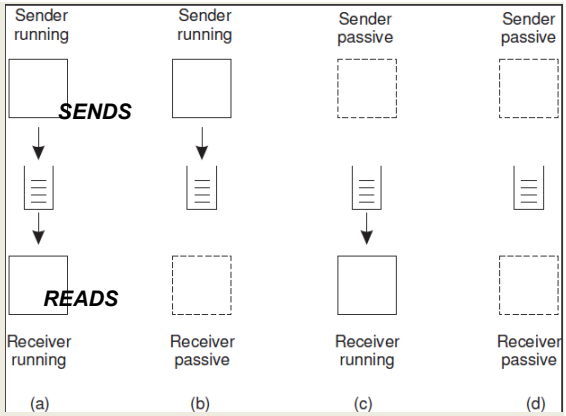


## MESSAGE QUEUEING SYSTEMS: USE CASES

- Enables communication between applications, or sets of processes
  - User applications
  - App-to-database
  - To support distributed real-time computations
- Use cases
  - Batch processing, Email, workflow, groupware, routing subqueries

## MESSAGE QUEUEING SYSTEMS

- Scenarios:
  (a) Sender/receiver both running
  (b) Sender running, receiver offline
  (c) Sender offline, receiver running
  (d) Sender/receiver both offline
- Queue persists msgs, and attempts to send them but no one may be available to receive them…






66

## MESSAGE QUEUEING SYSTEMS - 2

- **Key:** Truly persistent messaging
- Message queueing systems can persist messages for awhile and senders and receivers can be offline
- **Messages**
- Contain *any* data, may have size limit
- Are properly addressed, to a destination queue
- **Basic Inteface**
- PUT: called by sender to append msg to specified queue
- GET: blocking call to remove oldest msg from specified queue
  - Blocked if queue is empty
- POLL: Non-blocking, gets msg from specified queue



67

## MESSAGE QUEUEING SYSTEMS ARCHITECTURE

- **Basic Interface cont'd**
- NOTIFY: install a callback function, for when msg is placed into a queue. Notifies receivers

- **Queue managers**: manage individual message queues as a separate process/library
- Applications get/put messages only from ***local*** queues
- Queue manager and apps share local network
- **ISSUES:**
- How should we reference the destination queue?
- How should names be resolved (looked-up)?
  - Contact address (host, port) pairs
  - Local look-up tables can be stored at each queue manager

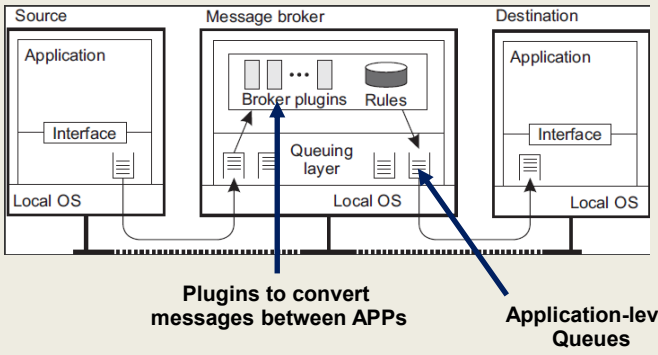


## MESSAGE QUEUEING SYSTEMS ARCHITECTURE - 2

- **ISSUES**:
- How do we route traffic between queue managers?
  - How are name-to-address mappings efficiently kept?
  - Each queue manager should be known to all others

- **Message brokers**
- Handle message conversion among different users/formats
- Addresses cases when senders and receivers don't speak the same protocol (language)
- Need arises for message protocol converters
  - "Reformatter" of messages
- Act as application-level gateway

## MESSAGE BROKER ORGANIZATION






70

## AMQP PROTOCOL

- Message-queueing systems initially developed to enable legacy applications to interoperate
- Decouple inter-application communication to "open" messaging-middleware
- Many are proprietary solutions, ***so not very open***
- e.g. Microsoft Message Queueing service, Windows NT 1997
- **Advanced message queueing protocol (AMQP)**, 2006
- Address openness/interoperability of proprietary solutions
- Open wire protocol for messaging with powerful routing capabilities
- Help *abstract* messaging and application interoperability by means of a generic open protocol
- Suffer from incompatibility among protocol versions
- pre-1.0, 1.0+



71

## AMQP - 2

- Consists of: Applications, Queue managers, Queues
- **Connections:** set up to a queue manager, TCP, with potentially many channels, stable, reused by many channels, long-lived
- **Channels:** support short-lived one-way communication
- **Sessions:** bi-directional communication across two channels
- **Link:** provide fine-grained flow-control of message transfer/status between applications and queue manager



## AMQP MESSAGING

- AMQP nodes: producer, consumer, queue
- Producer/consumer: represent regular applications
- Queues: store/forward messages

- Persistent messaging:
- **Messages** can be marked ***durable***
- These messages can only be delivered by nodes able to recover in case of failure
- Non-failure resistant nodes must reject durable messages
- **Source/target** nodes can be marked ***durable***
- Track what is durable (node state, node+msgs)

## MESSAGE-ORIENTED-MIDDLEWARE EXAMPLES:

- **Some examples:**
- RabbitMQ, Apache QPid
  - Implement Advanced Message Queueing Protocol (AMQP)
- Apache Kafka
  - **Dumb broker** (message store), similar to a distributed log file
  - **Smart consumers** – intelligence pushed off to the clients
  - Stores stream of records in categories called topics
  - Supports voluminous data, many consumers, with minimal O/H
  - Kafka **does not track** which messages were read by each consumer
  - Messages are removed after timeout
  - Clients must track their own consumption (*Kafka doesn't help*)
  - Messages have key, value, timestamp
  - Supports high volume pub/sub messaging and streams



74

## CH. 4.4: MULTICAST COMMUNICATION


Apache ActiveMQ



75

# CHAPTER 4

- 4.1 Foundations
  - Protocols
  - Types of communication
- 4.2 Remote procedure call
- 4.3 Message-oriented communication
  - Socket communication
  - Messaging libraries
  - Message-Passing Interface (MPI)
  - Message-queueing systems
  - Examples
- 4.4 Multicast communication
  - Flooding-based multicasting
  - Gossip-based data dissemination

**These sections feature many details, Our focus is on the "big picture"**



76

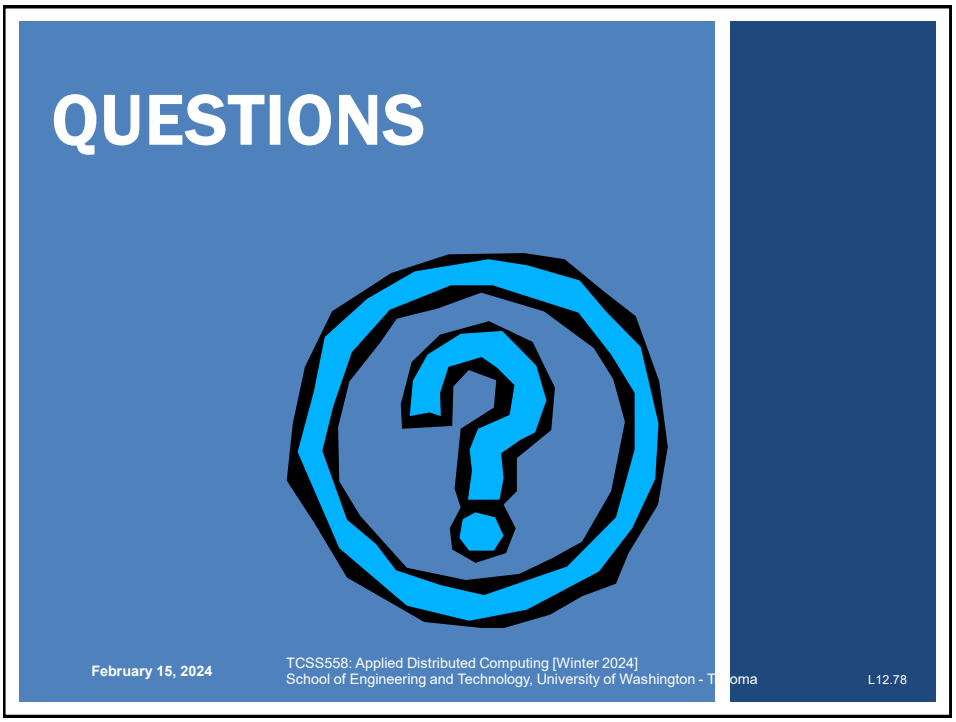
# MULTICAST COMMUNICATION

- Sending data to multiple receivers
- Many ***failed*** proposals for network-level / transport-level protocols to support multicast communication
- **Problem:** How to set up communication paths for information dissemination?
- **Solutions:** require huge management effort, human intervention

- Focus shifted more recently to **peer-to-peer** networks
  - Structured overlay networks can be setup easily and provide efficient communication paths
  - Application-level multicasting techniques more successful
  - Gossip-based dissemination: unstructured p2p networks



77


QUESTIONS
February 15, 2024
TCSS558: Applied Distributed Computing [Winter 2024]
School of Engineering and Technology, University of Washington - Tacoma
L12.78

78