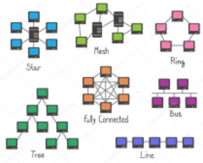
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Ch. 4 – Communication - II

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



1

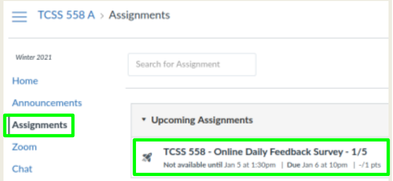
OBJECTIVES – 2/21

- Questions from 2/16
- Assignment 1: Key Value Store
- Assignment 2: Replicated Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-overview)
  - Chapter 4.3: Message Oriented Communication
  - Chapter 4.4: Multicast 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



TCSS 558 A › Assignments

Winter 2021

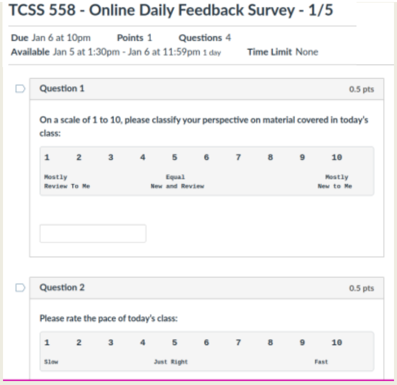
Search for Assignment

Home
Announcements
Assignments
Zoom
Chat

▾ Upcoming Assignments

TCSS 558 - Online Daily Feedback Survey - 1/5
Not available until Jan 5 at 1:30pm | Due Jan 6 at 10pm | -/1 pts

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                          Equal                                       Mostly
Review To Me               New and Review                                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 (~31 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- <u>**Average – 6.03 (↓ - *previous 6.70*)**</u>

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- <u>**Average – 5.34 (↓ - *previous 5.61*)**</u>

5

TENURE TRACK FACULTY CANDIDATE RESEARCH SEMINARS – EXTRA CREDIT

- Tuesday February 21 – 12:30pm – Cherry Parkes room 106
- Thursday February 23 – 12:30pm – Cherry Parkes room 106
- Wednesday March 1 – 12:30pm – Cherry Parkes room TBA
- Friday March 3 – 12:30pm – Cherry Parkes room TBA

- Earn up to 2.5% extra credit added to the overall course grade
- Scored out of 5 total points
- First seminar – earn 2 points
- 2nd, 3rd, 4th seminar – earn 1 point each

Add to final course grade:
(Total_seminar_points / 5 points) * 2.5%

6

FEEDBACK FROM 2/16

- ***For the mid-term question on shared data space implementation with an unstructured peer-to-peer architecture:***
- ***Is availability/accessibility both an advantage and disadvantage ?***
- ADVANTAGE: The unstructured peer-to-peer implementation features multiple nodes. This enables requests to be distributed across multiple nodes better. The centralized implementation is not really a "distributed system". It can't really scale, and will be maxed-out if there are too many client requests.
- DISADVANTAGE: As opposed to accessibility/availability, the disadvantage is Failure Transparency. If one node containing a subset of data is slow or unavailable it is very difficult to tell with unstructured peer-to-peer. It is easier to tell that the central server has failed. Either ALL or NONE of the data will be available with central server. Small amounts of data could become inaccessible and it may be difficult for a user to discern this. The system remains accessible/available, just some data is missing !

OBJECTIVES – 2/21

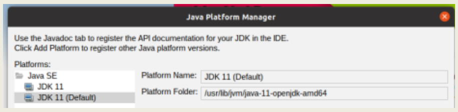
- Questions from 2/16
- **Assignment 1: Key Value Store**
- Assignment 2: Replicated Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-overview)
  - Chapter 4.3: Message Oriented Communication
  - Chapter 4.4: Multicast Communication

ASSIGNMENT 1

- Team 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

USING JAVA 11 IN NETBEANS

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



Use the Javadoc tab to register the API documentation for your JDK in the IDE.
Click Add Platform to register other Java platform versions.

Platforms:
Java SE
JDK 11
JDK 11 (Default)

Platform Name: JDK 11 (Default)
Platform Folder: /usr/lib/jvm/java-11-openjdk-amd64

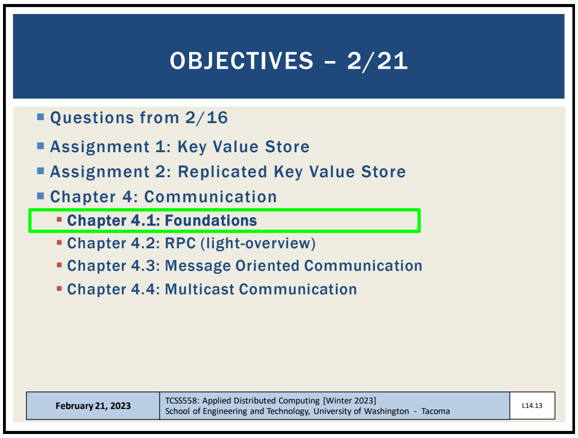
- On left-hand Project menu, right-click on '**GenericNode**' project
- Select Properties
- Under Build | Compile, be sure Java Platform is JDK 11
- Under Sources, be sure Source/Binary Format is 11

OBJECTIVES – 2/21

- **Questions from 2/16**
- Assignment 1: Key Value Store
- **Assignment 2: Replicated Key Value Store**
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-overview)
  - Chapter 4.3: Message Oriented Communication
  - Chapter 4.4: Multicast Communication

ASSIGNMENT 2

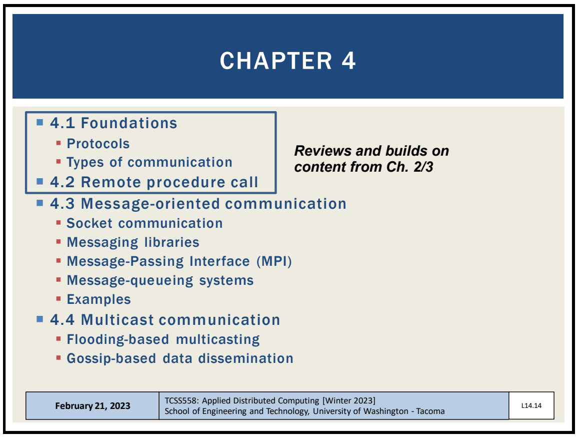
- **Sunday March 12th**
- Goal: Replicated Key Value Store
- Team signup to be posted on Canvas under 'People'
- Builds off of Assignment 1 GenericNode
- Focus on TCP client/server w/ replication
- How to track membership for data replication?
  - Can implement multiple types of membership tracking for extra credit



OBJECTIVES – 2/21

- Questions from 2/16
- Assignment 1: Key Value Store
- Assignment 2: Replicated Key Value Store
- Chapter 4: Communication
  - **Chapter 4.1: Foundations**
  - Chapter 4.2: RPC (light-overview)
  - Chapter 4.3: Message Oriented Communication
  - Chapter 4.4: Multicast Communication

13

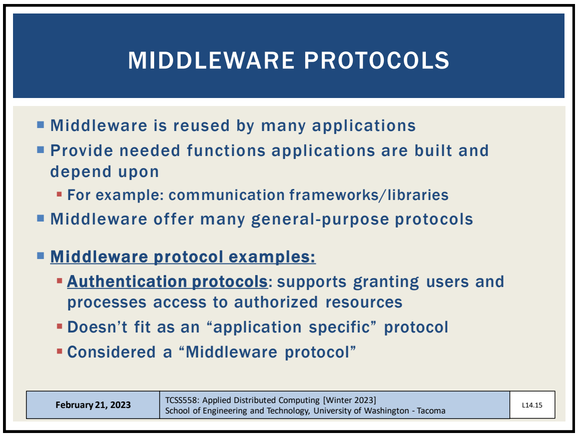


CHAPTER 4

- 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

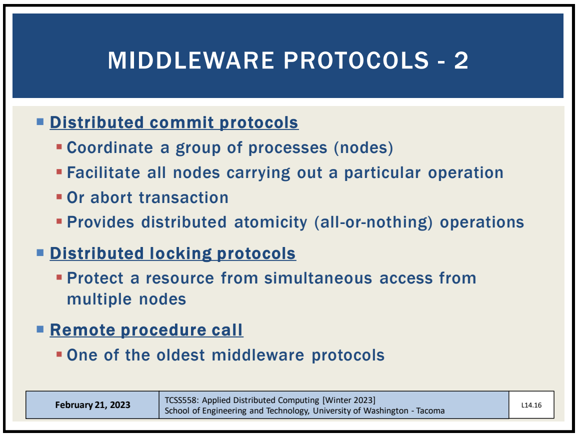
14



MIDDLEWARE PROTOCOLS

- Middleware is reused by many applications
- Provide needed functions applications are built and depend upon
  - For example: communication frameworks/libraries
- Middleware offer many general-purpose protocols
- **Middleware protocol examples:**
  - **Authentication protocols**: supports granting users and processes access to authorized resources
  - Doesn't fit as an "application specific" protocol
  - Considered a "Middleware protocol"

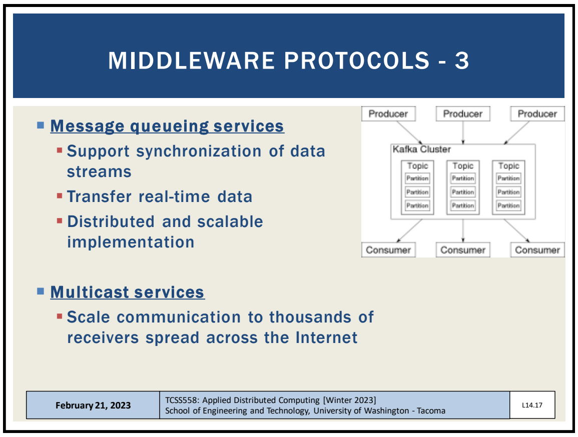
15



MIDDLEWARE PROTOCOLS - 2

- **Distributed commit protocols**
  - Coordinate a group of processes (nodes)
  - Facilitate all nodes carrying out a particular operation
  - Or abort transaction
  - Provides distributed atomicity (all-or-nothing) operations
- **Distributed locking protocols**
  - Protect a resource from simultaneous access from multiple nodes
- **Remote procedure call**
  - One of the oldest middleware protocols

16

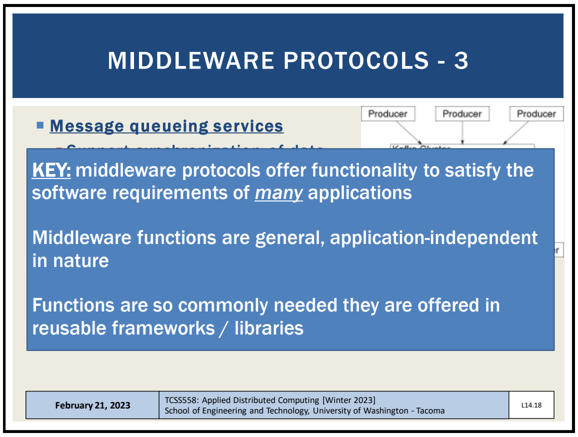


MIDDLEWARE PROTOCOLS - 3

- **Message queueing services**
  - Support synchronization of data streams
  - Transfer real-time data
  - Distributed and scalable implementation
- **Multicast services**
  - Scale communication to thousands of receivers spread across the Internet

Producer
Producer
Producer
Kafka Cluster
Topic
Partition
Partition
Partition
Topic
Partition
Partition
Partition
Topic
Partition
Partition
Partition
Consumer
Consumer
Consumer

17



MIDDLEWARE PROTOCOLS - 3

- **Message queueing services**

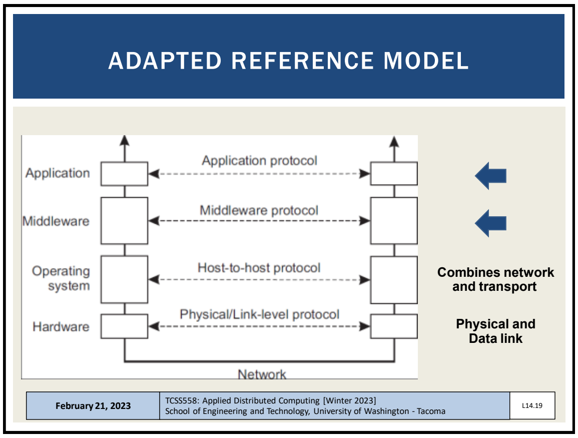
Producer
Producer
Producer

KEY: middleware protocols offer functionality to satisfy the software requirements of *many* applications

Middleware functions are general, application-independent in nature

Functions are so commonly needed they are offered in reusable frameworks / libraries

18



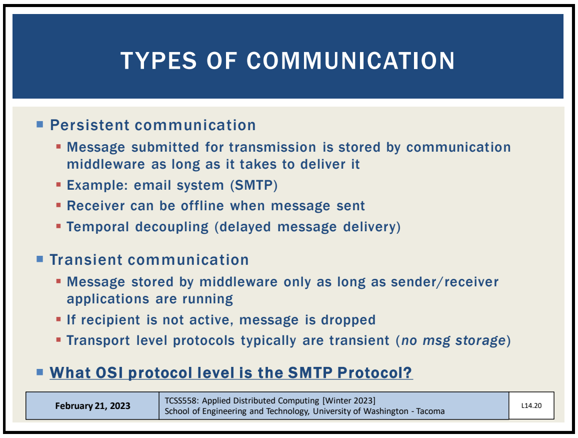
ADAPTED REFERENCE MODEL

Combines network and transport

Physical and Data link

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.19

19

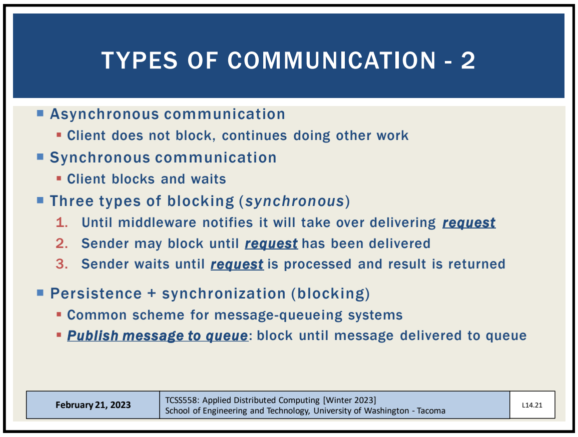


TYPES OF COMMUNICATION

- 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?**

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.20

20

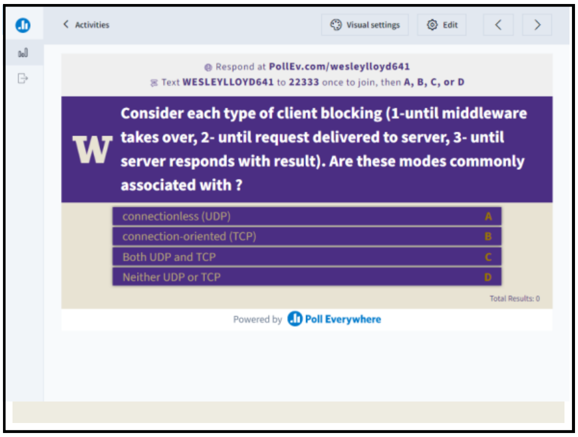


TYPES OF COMMUNICATION - 2

- Asynchronous communication
  - Client does not block, continues doing other work
- Synchronous communication
  - Client blocks and waits
- Three types of blocking (*synchronous*)
  1. Until middleware notifies it will take over delivering ***request***
  2. Sender may block until ***request*** has been delivered
  3. 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

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.21

21



Respond at **PollEv.com/wesleylloyd641**

Text **WESLEYLLOYD641** to **22333** once to join, then **A, B, C, or D**

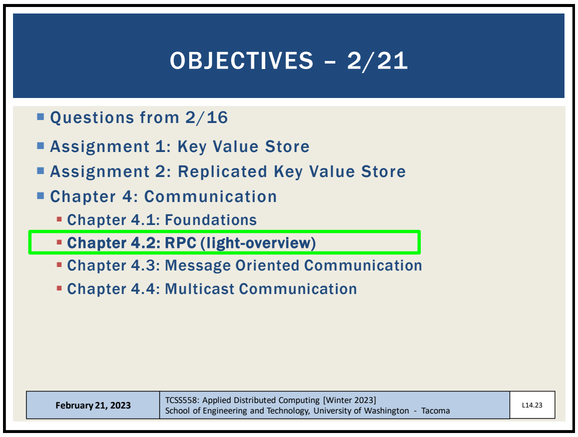
Consider each type of client blocking (1-until middleware takes over, 2- until request delivered to server, 3- until server responds with result). Are these modes commonly associated with ?

- connectionless (UDP) A
- connection-oriented (TCP) B
- Both UDP and TCP C
- Neither UDP or TCP D

Total Results: 0

Powered by Poll Everywhere

22

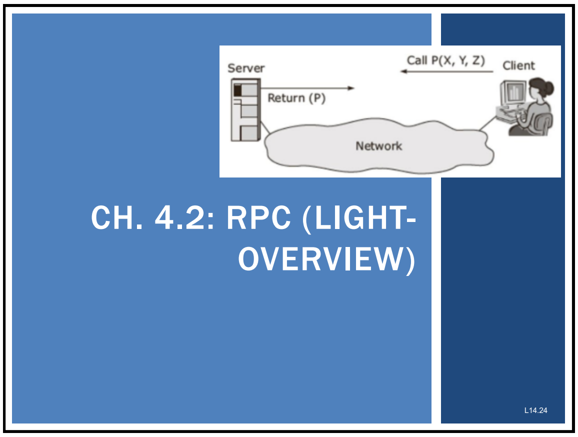


OBJECTIVES – 2/21

- Questions from 2/16
- Assignment 1: Key Value Store
- Assignment 2: Replicated Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - **Chapter 4.2: RPC (light-overview)**
  - Chapter 4.3: Message Oriented Communication
  - Chapter 4.4: Multicast Communication

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.23

23



CH. 4.2: RPC (LIGHT-OVERVIEW)

L14.24

24

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

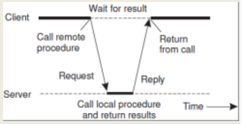
These sections feature many details,
Our focus is on the "big picture"

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)`

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***



RPC - 3

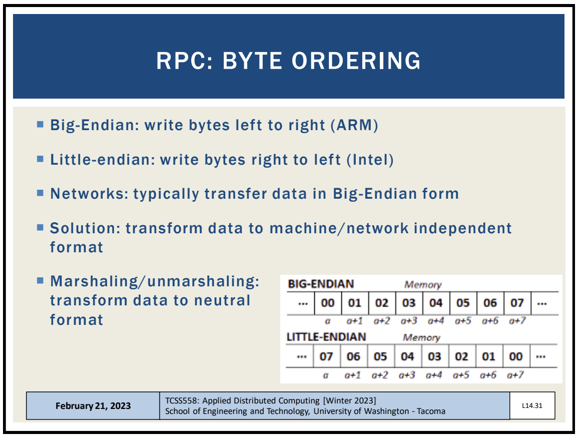
- Server packs procedure ***results*** and sends back to client.
- Client "***request***" call unblocks and data is unpacked
- Client can't tell method was called remotely over the network… ***except for network latency…***
- Call abstraction enables clients to invoke functions in alternate languages,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

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

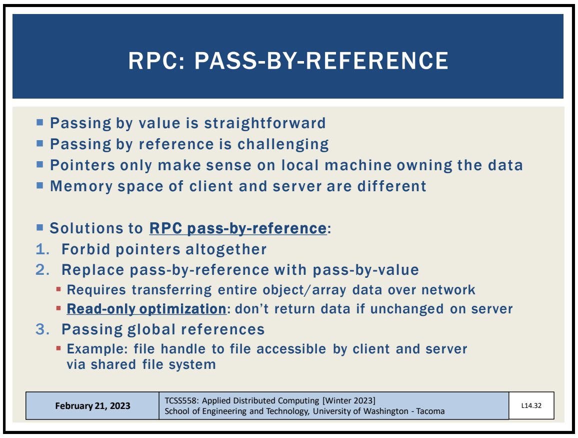
BIG-ENDIAN — *Memory*

…	00	01	02	03	04	05	06	07	…
	a	*a+1*	*a+2*	*a+3*	*a+4*	*a+5*	*a+6*	*a+7*	

LITTLE-ENDIAN — *Memory*

…	07	06	05	04	03	02	01	00	…
	a	*a+1*	*a+2*	*a+3*	*a+4*	*a+5*	*a+6*	*a+7*	

31



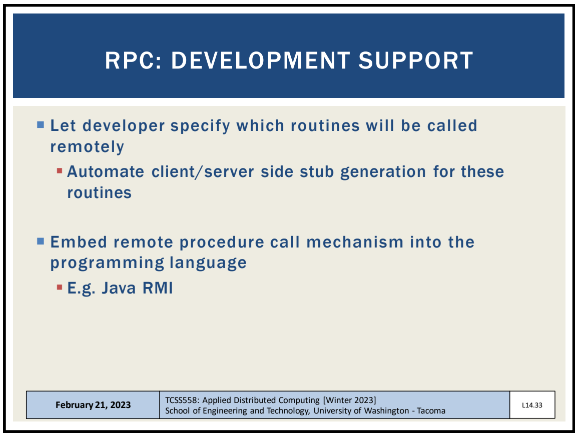
RPC: PASS-BY-REFERENCE

- 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

32

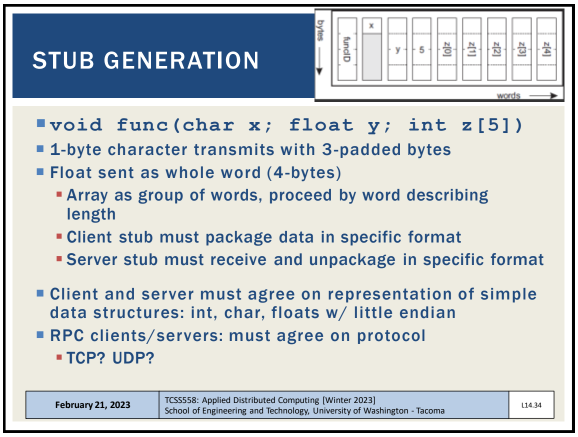


RPC: DEVELOPMENT SUPPORT

- 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

33



STUB GENERATION

bytes

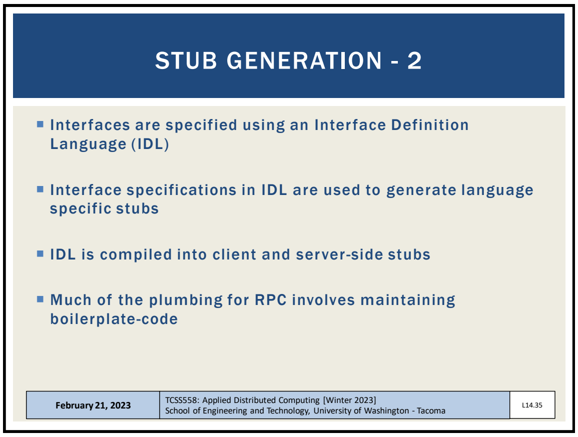
funcID	x	y	5	z[0]	z[1]	z[2]	z[3]	z[4]

words

- `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 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?

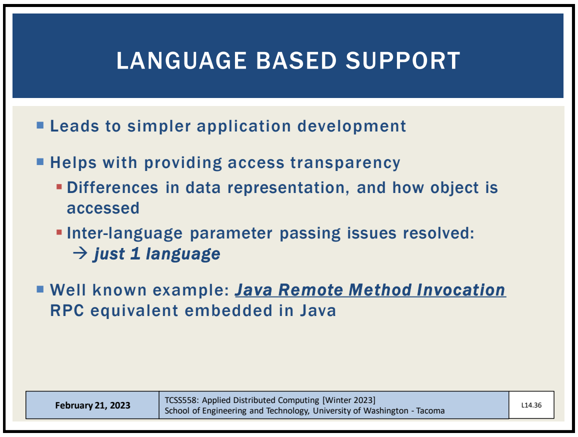
34



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

35

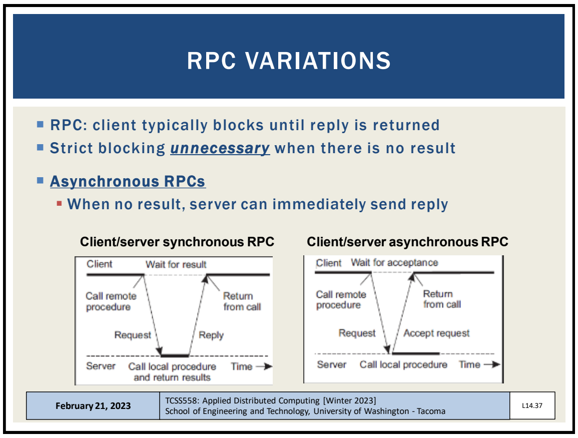


LANGUAGE BASED SUPPORT

- 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

36



RPC VARIATIONS

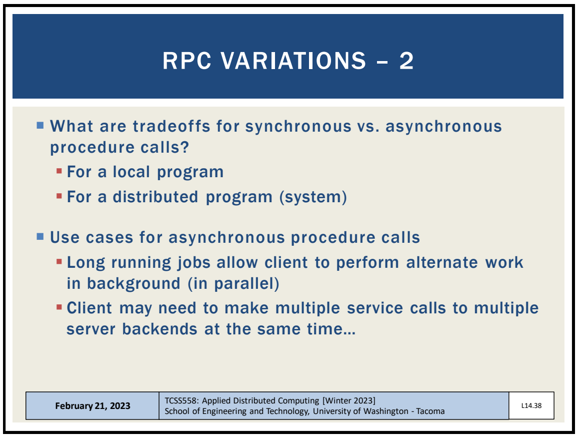
- RPC: client typically blocks until reply is returned
- Strict blocking ***unnecessary*** when there is no result
- **Asynchronous RPCs**
  - When no result, server can immediately send reply

Client/server synchronous RPC

Client/server asynchronous RPC

Client
Wait for result
Call remote procedure
Return from call
Request
Reply
Server
Call local procedure and return results
Time
Client
Wait for acceptance
Call remote procedure
Return from call
Request
Accept request
Server
Call local procedure
Time

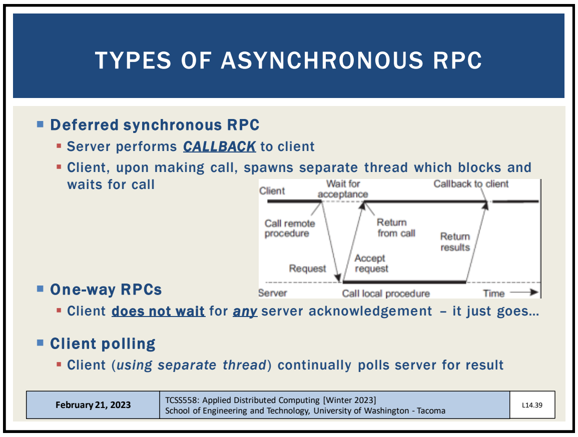
37



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…

38



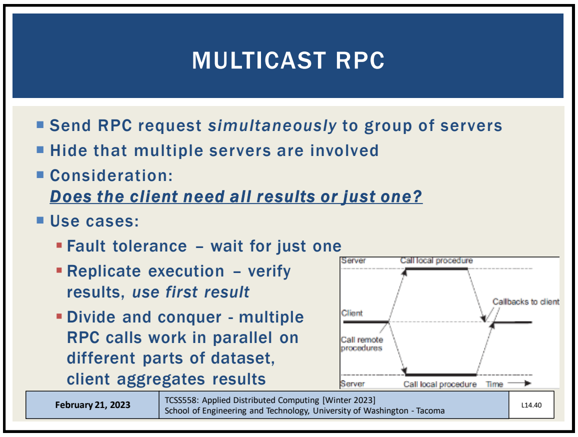
TYPES OF ASYNCHRONOUS RPC

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

Client
Wait for acceptance
Callback to client
Call remote procedure
Return from call
Return results
Accept request
Request
Server
Call local procedure
Time

- **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

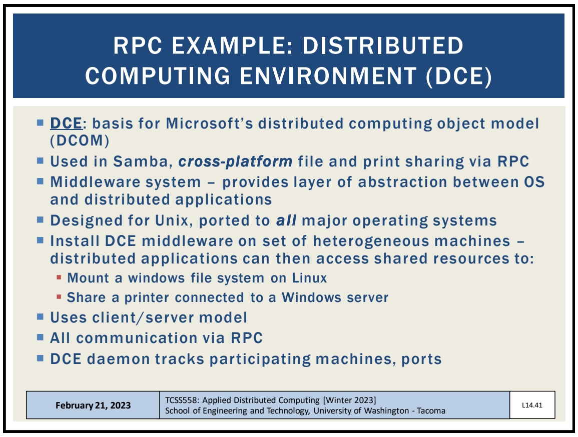
39



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

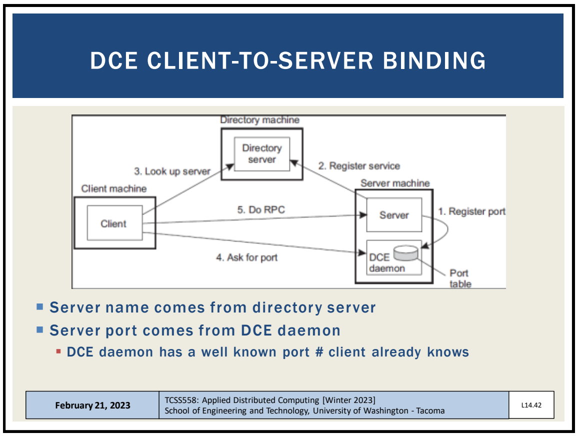
40



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

41

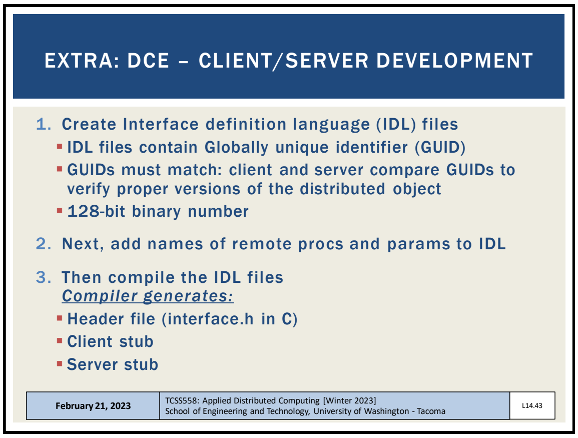


DCE CLIENT-TO-SERVER BINDING

Directory machine
Directory server
3. Look up server
2. Register service
Server machine
Client machine
5. Do RPC
Client
Server
1. Register port
4. Ask for port
DCE daemon
Port table

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

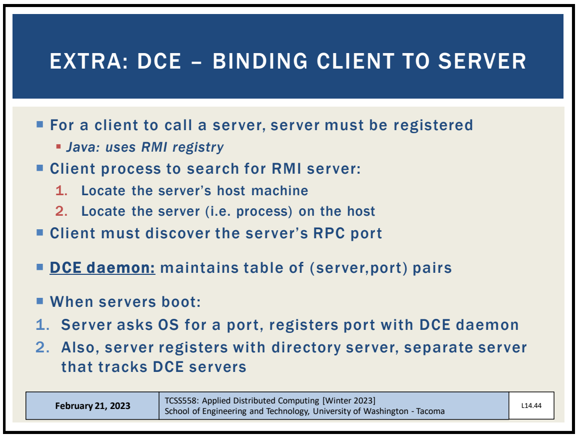
42



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

43



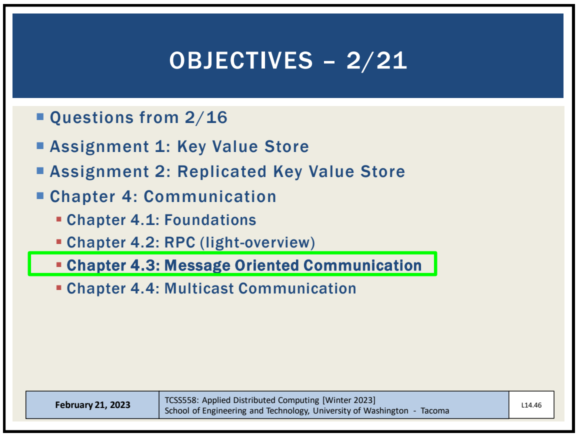
EXTRA: DCE – BINDING CLIENT TO SERVER

- 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

44

WE WILL RETURN AT 2:40 PM

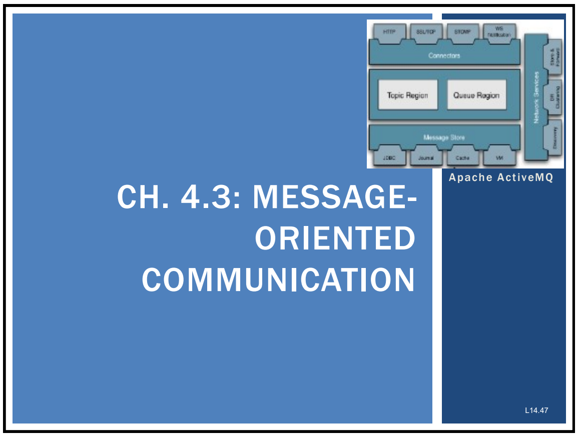
45



OBJECTIVES – 2/21

- Questions from 2/16
- Assignment 1: Key Value Store
- Assignment 2: Replicated Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-overview)
  - **Chapter 4.3: Message Oriented Communication**
  - Chapter 4.4: Multicast Communication

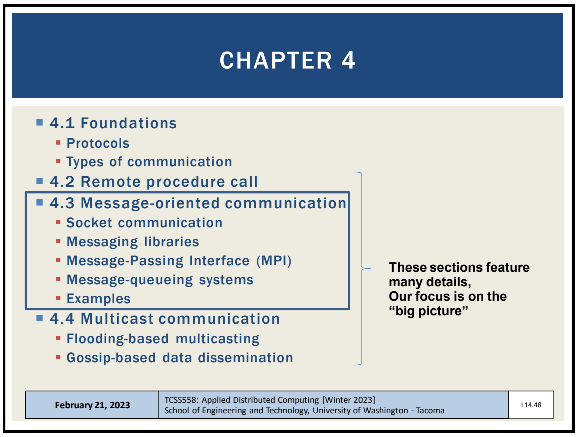
46



Apache ActiveMQ

CH. 4.3: MESSAGE-ORIENTED COMMUNICATION

47



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"

48

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

49

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

50

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

51

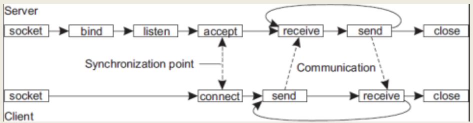
SERVER SOCKET OPERATIONS

- **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

52

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

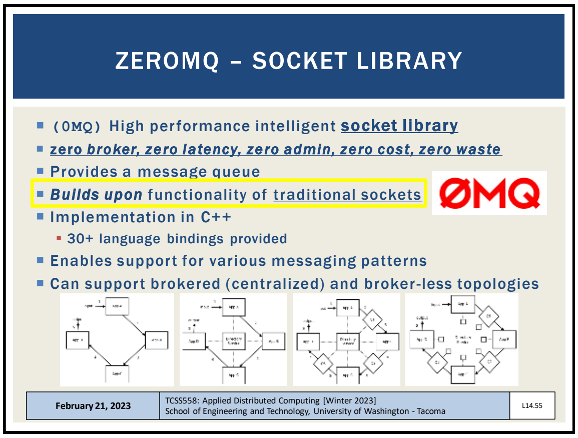


53

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

54

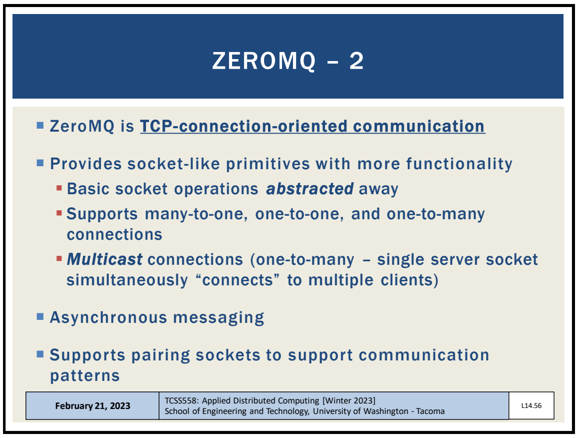


ZEROMQ – SOCKET LIBRARY

- (ØMQ) 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

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.55

55

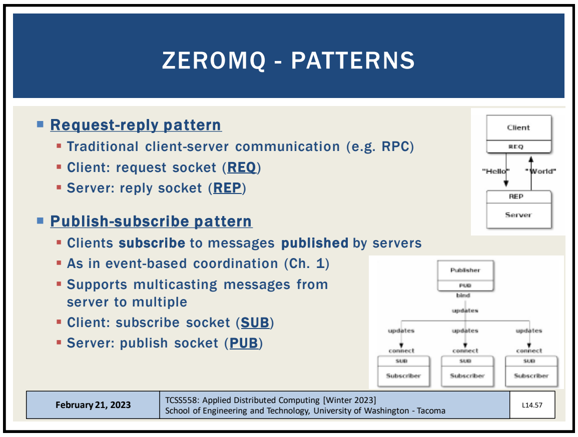


ZEROMQ – 2

- 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

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.56

56

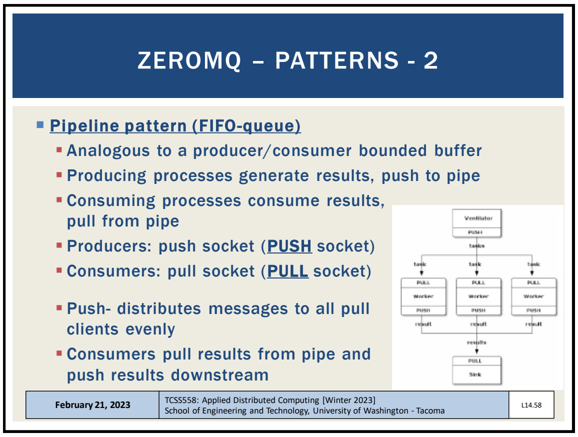


ZEROMQ - PATTERNS

- **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**)

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.57

57

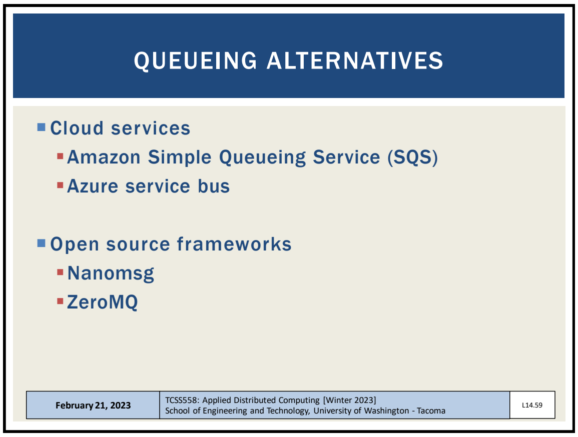


ZEROMQ – PATTERNS - 2

- **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

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.58

58

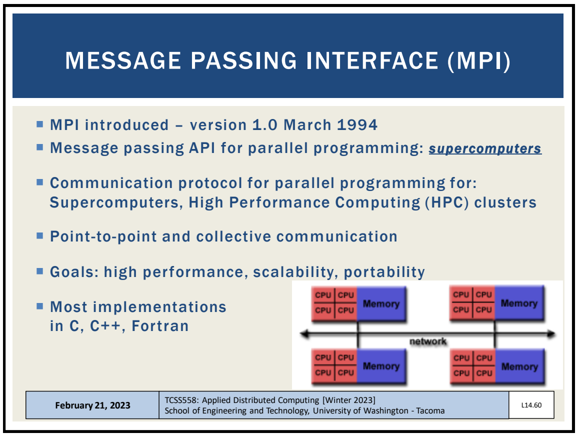


QUEUEING ALTERNATIVES

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

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.59

59

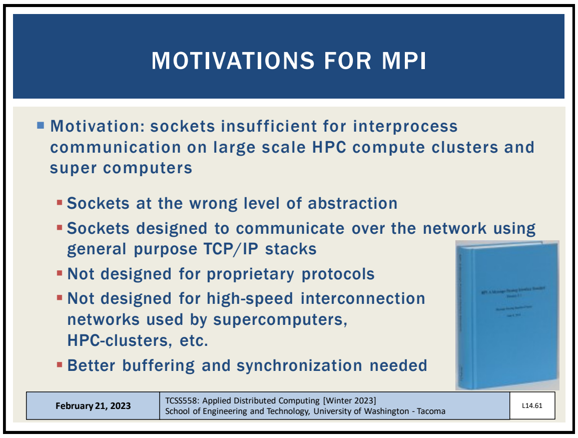


MESSAGE PASSING INTERFACE (MPI)

- 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

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.60

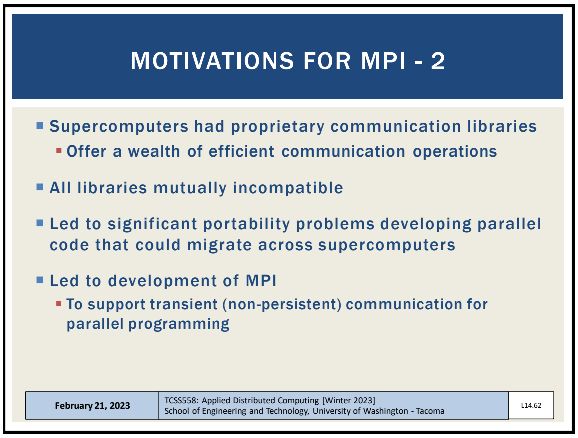
60



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

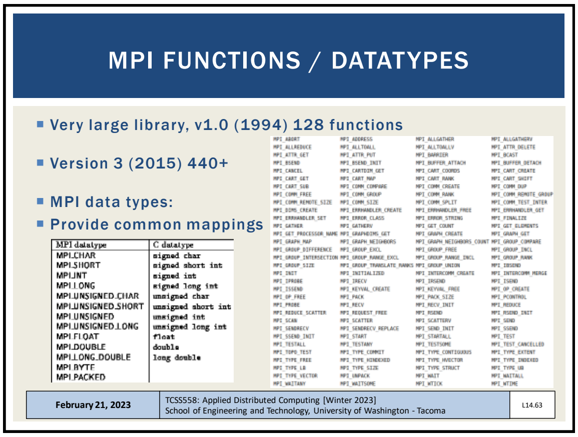
61



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

62



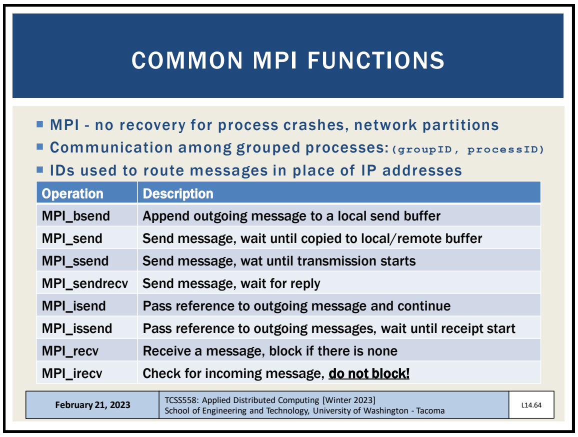
MPI FUNCTIONS / DATATYPES

- 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
```

63

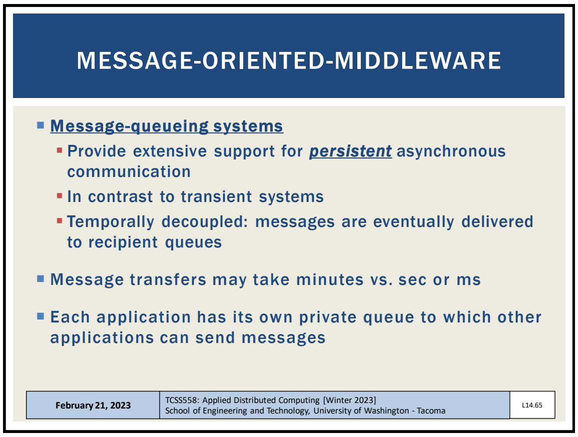


COMMON MPI FUNCTIONS

- 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, wait 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!**

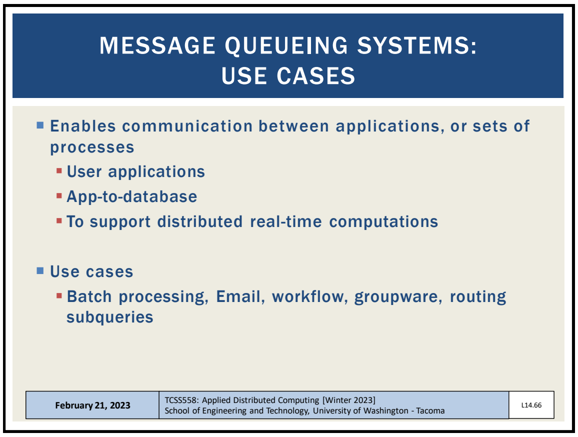
64



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

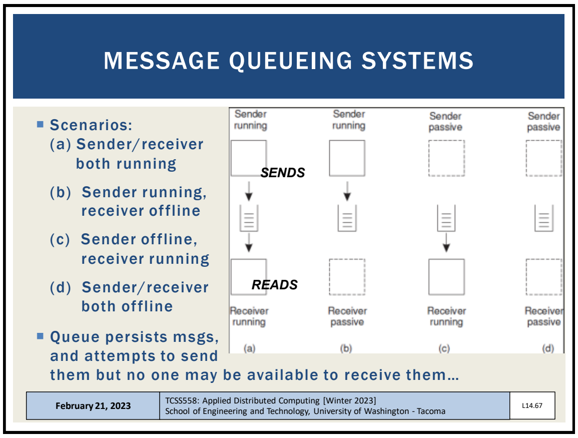
65



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

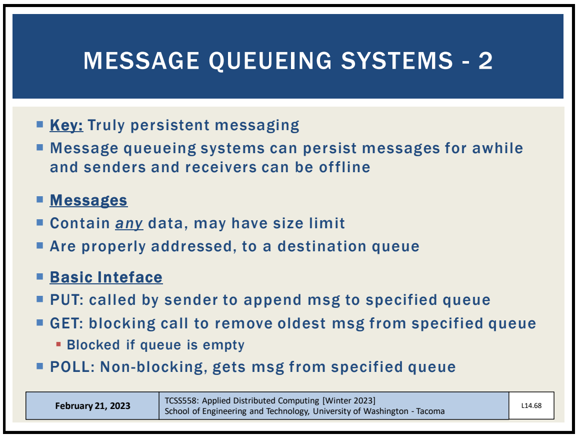
66



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…

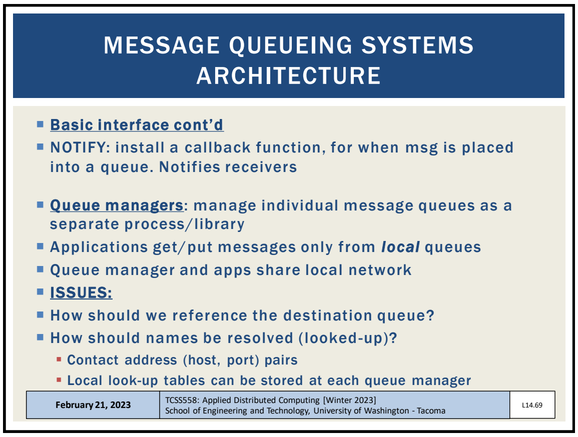
67



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

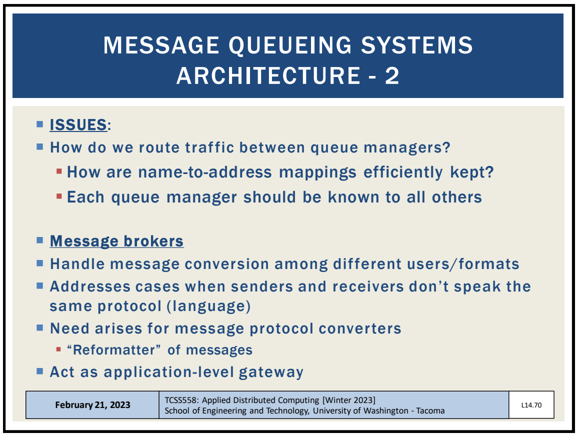
68



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

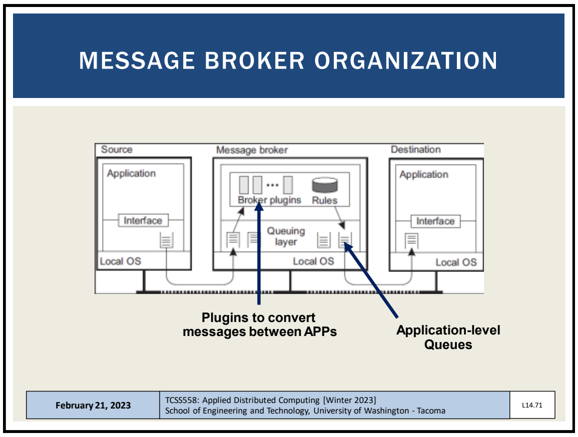
69



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

70

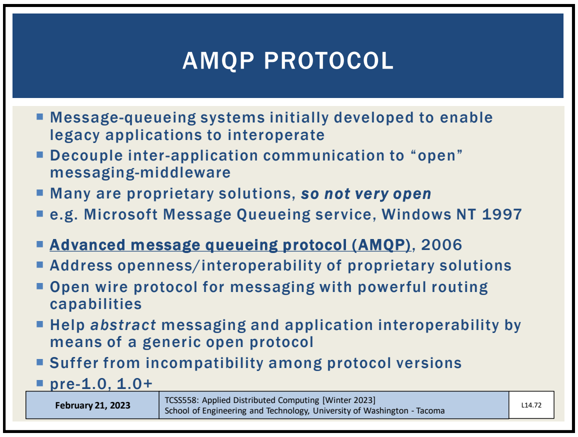


MESSAGE BROKER ORGANIZATION

Plugins to convert messages between APPs

Application-level Queues

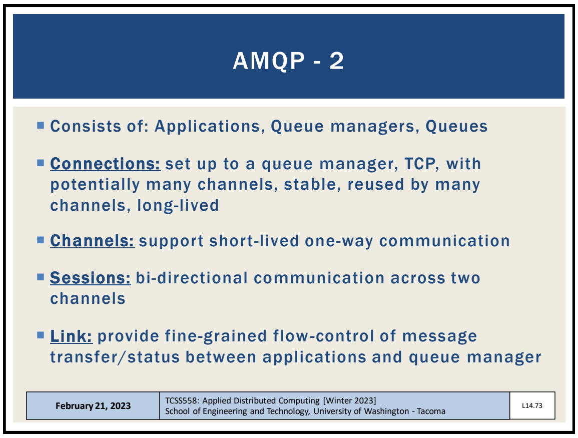
71



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+

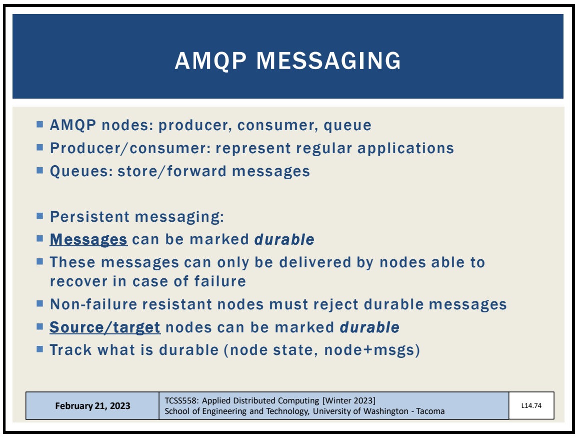
72



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

73

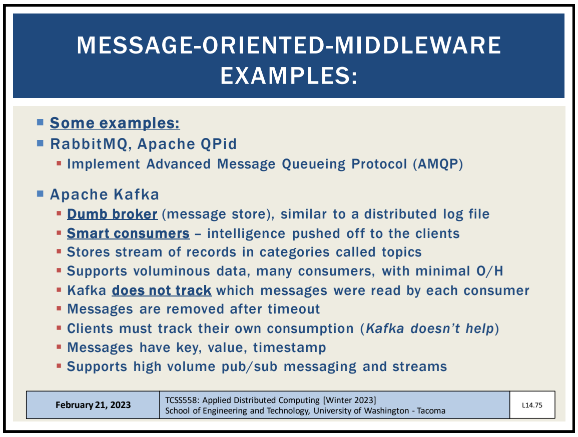


AMQP MESSAGING

- 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)

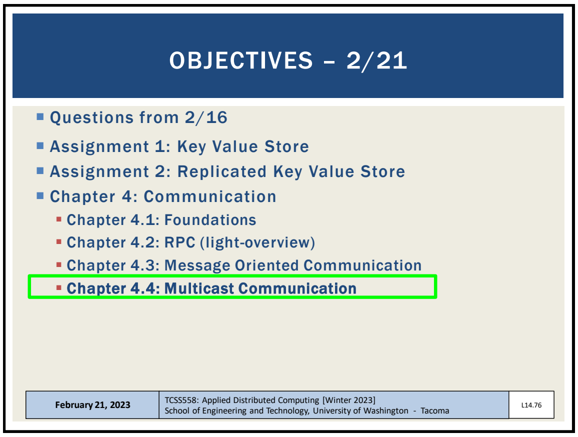
74



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

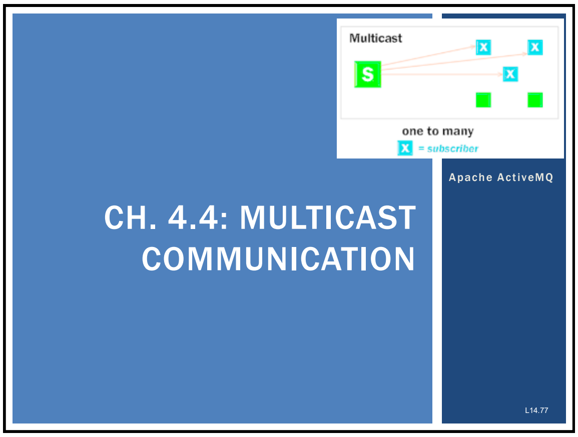
75



OBJECTIVES – 2/21

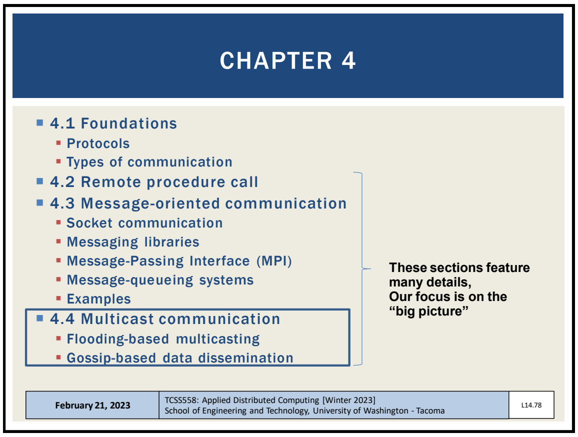
- Questions from 2/16
- Assignment 1: Key Value Store
- Assignment 2: Replicated Key Value Store
- Chapter 4: Communication
  - Chapter 4.1: Foundations
  - Chapter 4.2: RPC (light-overview)
  - Chapter 4.3: Message Oriented Communication
  - **Chapter 4.4: Multicast Communication**

76


Multicast
S
X
X
X
one to many
X = subscriber
Apache ActiveMQ

CH. 4.4: MULTICAST COMMUNICATION

77

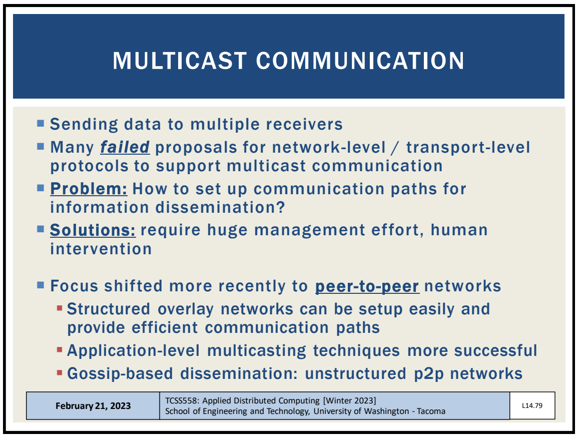


CHAPTER 4

- 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"

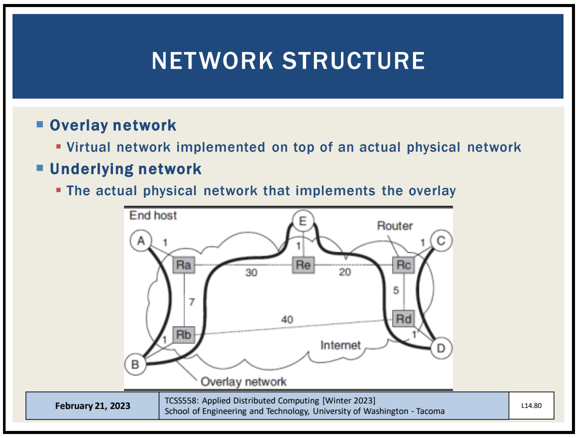
78



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

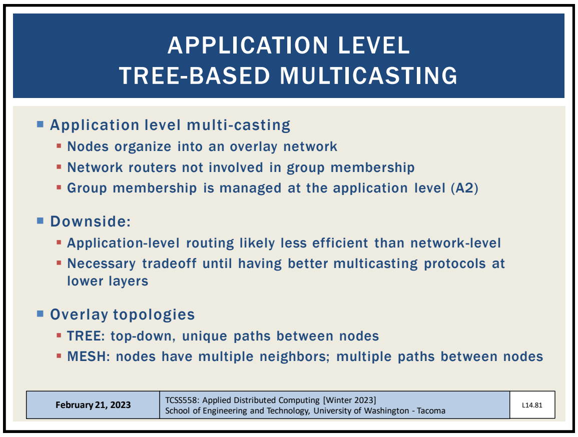
79



NETWORK STRUCTURE

- **Overlay network**
  - Virtual network implemented on top of an actual physical network
- **Underlying network**
  - The actual physical network that implements the overlay

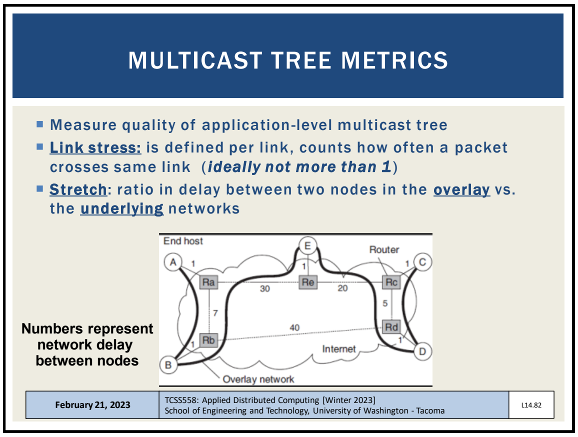
80



APPLICATION LEVEL TREE-BASED MULTICASTING

- Application level multi-casting
  - Nodes organize into an overlay network
  - Network routers not involved in group membership
  - Group membership is managed at the application level (A2)
- Downside:
  - Application-level routing likely less efficient than network-level
  - Necessary tradeoff until having better multicasting protocols at lower layers
- Overlay topologies
  - TREE: top-down, unique paths between nodes
  - MESH: nodes have multiple neighbors; multiple paths between nodes

81

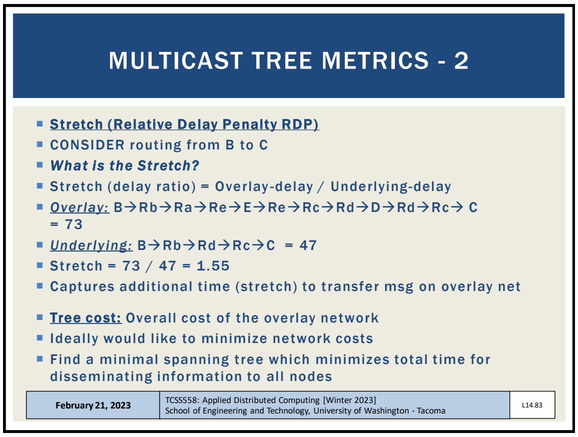


MULTICAST TREE METRICS

- Measure quality of application-level multicast tree
- **Link stress:** is defined per link, counts how often a packet crosses same link (***ideally not more than 1***)
- **Stretch**: ratio in delay between two nodes in the **overlay** vs. the **underlying** networks

Numbers represent network delay between nodes

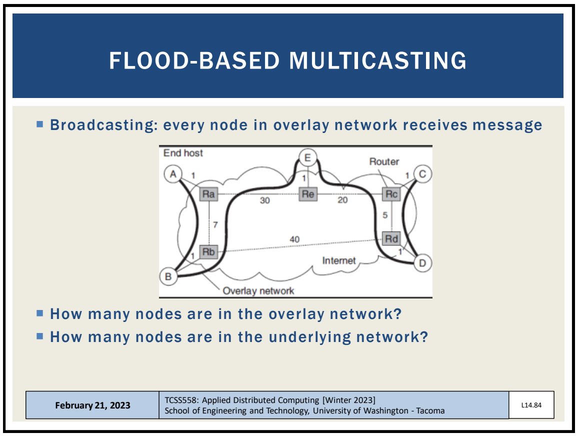
82



MULTICAST TREE METRICS - 2

- **Stretch (Relative Delay Penalty RDP)**
- CONSIDER routing from B to C
- ***What is the Stretch?***
- Stretch (delay ratio) = Overlay-delay / Underlying-delay
- *Overlay:* B→Rb→Ra→Re→E→Re→Rc→Rd→D→Rd→Rc→ C = 73
- *Underlying:* B→Rb→Rd→Rc→C = 47
- Stretch = 73 / 47 = 1.55
- Captures additional time (stretch) to transfer msg on overlay net
- **Tree cost:** Overall cost of the overlay network
- Ideally would like to minimize network costs
- Find a minimal spanning tree which minimizes total time for disseminating information to all nodes

83

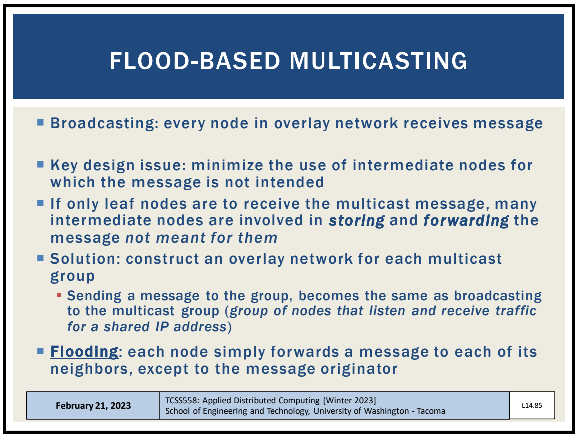


FLOOD-BASED MULTICASTING

- Broadcasting: every node in overlay network receives message

- How many nodes are in the overlay network?
- How many nodes are in the underlying network?

84

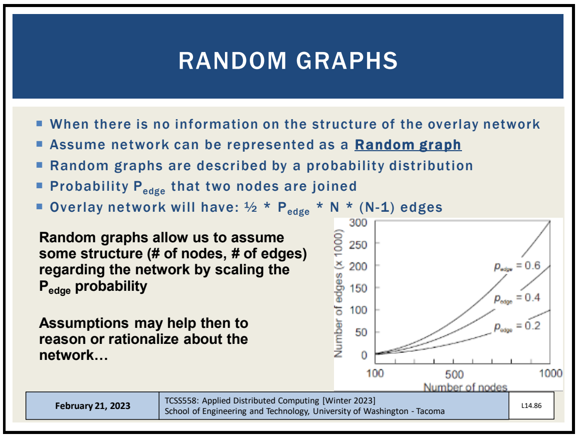


FLOOD-BASED MULTICASTING

- Broadcasting: every node in overlay network receives message
- Key design issue: minimize the use of intermediate nodes for which the message is not intended
- If only leaf nodes are to receive the multicast message, many intermediate nodes are involved in ***storing*** and ***forwarding*** the message *not meant for them*
- Solution: construct an overlay network for each multicast group
  - Sending a message to the group, becomes the same as broadcasting to the multicast group (*group of nodes that listen and receive traffic for a shared IP address*)
- **Flooding**: each node simply forwards a message to each of its neighbors, except to the message originator

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.85

85



RANDOM GRAPHS

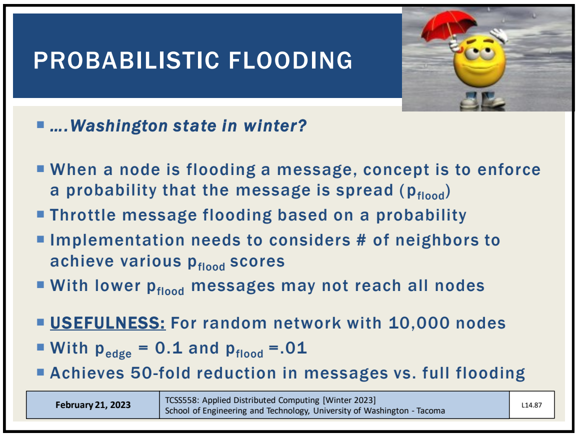
- When there is no information on the structure of the overlay network
- Assume network can be represented as a **Random graph**
- Random graphs are described by a probability distribution
- Probability P_{edge} that two nodes are joined
- Overlay network will have: ½ * P_{edge} * N * (N-1) edges

Random graphs allow us to assume some structure (# of nodes, # of edges) regarding the network by scaling the P_{edge} probability

Assumptions may help then to reason or rationalize about the network…

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.86

86

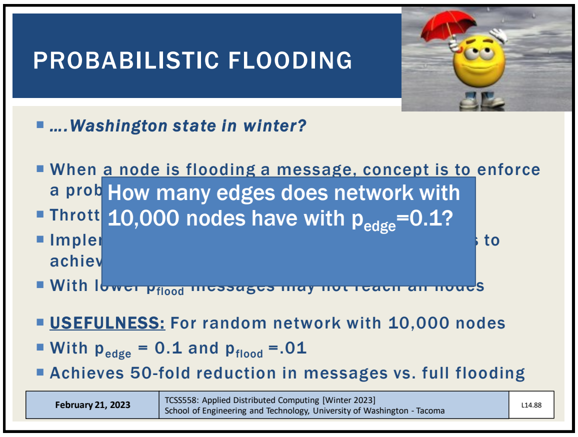


PROBABILISTIC FLOODING

- ***….Washington state in winter?***
- When a node is flooding a message, concept is to enforce a probability that the message is spread (p_{flood})
- Throttle message flooding based on a probability
- Implementation needs to considers # of neighbors to achieve various p_{flood} scores
- With lower p_{flood} messages may not reach all nodes
- **USEFULNESS:** For random network with 10,000 nodes
- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.87

87



PROBABILISTIC FLOODING

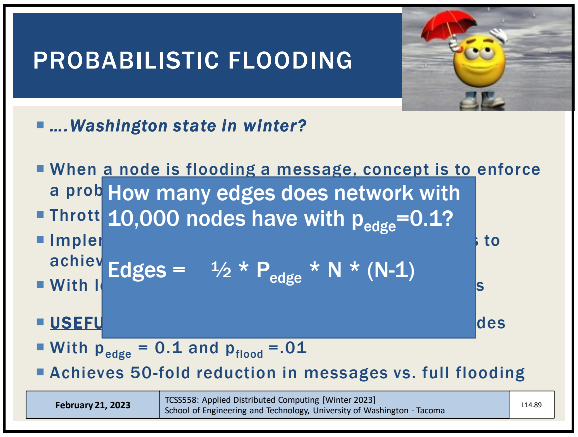
- ***….Washington state in winter?***
- When a node is flooding a message, concept is to enforce a prob
- Thrott
- Impler s to achiev
- With lower p_{flood} messages may not reach all nodes

How many edges does network with 10,000 nodes have with p_{edge}=0.1?

- **USEFULNESS:** For random network with 10,000 nodes
- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.88

88



PROBABILISTIC FLOODING

- ***….Washington state in winter?***
- When a node is flooding a message, concept is to enforce a prob
- Thrott
- Impler s to achiev
- With l s
- **USEFU** des

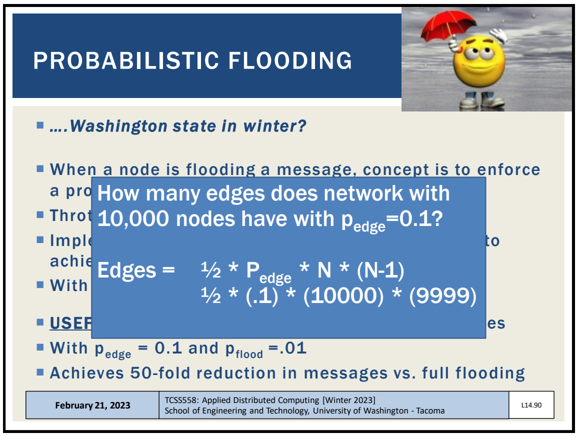
How many edges does network with 10,000 nodes have with p_{edge}=0.1?

Edges = ½ * P_{edge} * N * (N-1)

- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.89

89



PROBABILISTIC FLOODING

- ***….Washington state in winter?***
- When a node is flooding a message, concept is to enforce a pro
- Throt
- Imple to achie
- With
- **USEF** es

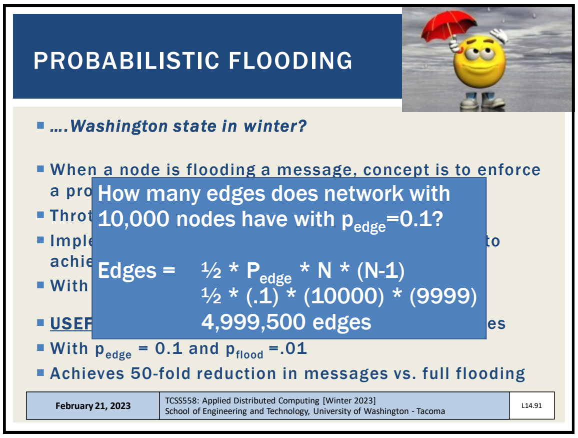
How many edges does network with 10,000 nodes have with p_{edge}=0.1?

Edges = ½ * P_{edge} * N * (N-1)
½ * (.1) * (10000) * (9999)

- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.90

90



PROBABILISTIC FLOODING

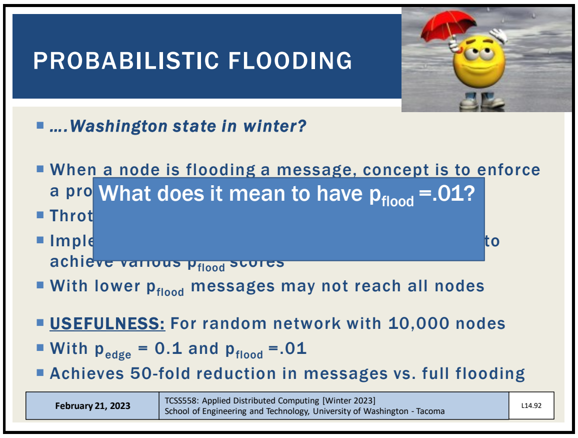
- ….Washington state in winter?
- When a node is flooding a message, concept is to enforce a pro
- Throt
- Imple … to achie
- With
- USEF … es
- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

How many edges does network with 10,000 nodes have with p_{edge}=0.1?

Edges = ½ * P_{edge} * N * (N-1)
½ * (.1) * (10000) * (9999)
4,999,500 edges

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.91

91



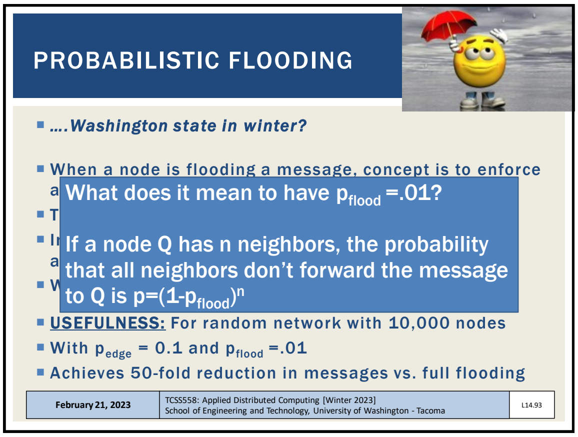
PROBABILISTIC FLOODING

- ….Washington state in winter?
- When a node is flooding a message, concept is to enforce a pro
- Throt
- Imple … to achieve various p_{flood} scores
- With lower p_{flood} messages may not reach all nodes
- USEFULNESS: For random network with 10,000 nodes
- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

What does it mean to have p_{flood} =.01?

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.92

92



PROBABILISTIC FLOODING

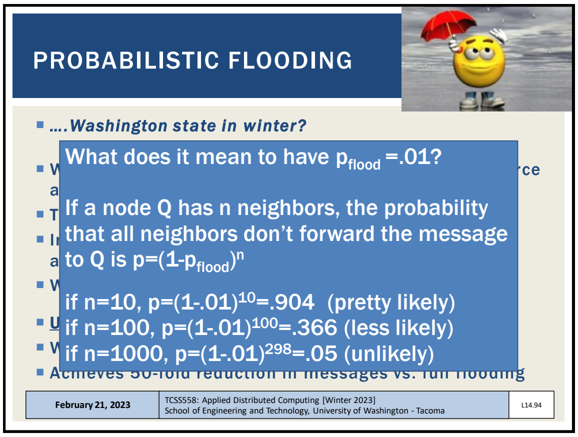
- ….Washington state in winter?
- When a node is flooding a message, concept is to enforce a
- T
- I a
- W
- USEFULNESS: For random network with 10,000 nodes
- With p_{edge} = 0.1 and p_{flood} =.01
- Achieves 50-fold reduction in messages vs. full flooding

What does it mean to have p_{flood} =.01?

If a node Q has n neighbors, the probability that all neighbors don't forward the message to Q is $p=(1-p_{flood})^n$

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.93

93



PROBABILISTIC FLOODING

- ….Washington state in winter?
- W … rce a
- T
- I a
- W
- U
- W
- Achieves 50-fold reduction in messages vs. full flooding

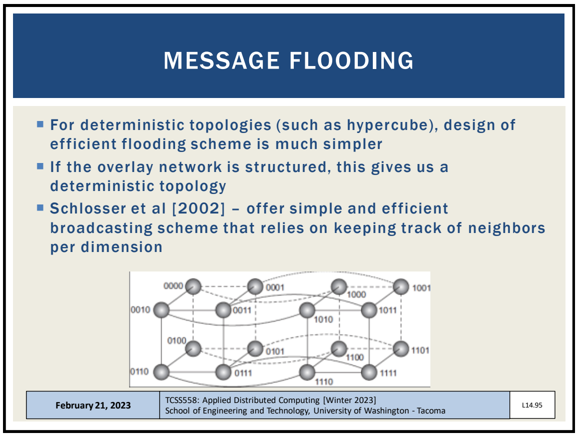
What does it mean to have p_{flood} =.01?

If a node Q has n neighbors, the probability that all neighbors don't forward the message to Q is $p=(1-p_{flood})^n$

if n=10, $p=(1-.01)^{10}=.904$ (pretty likely)
if n=100, $p=(1-.01)^{100}=.366$ (less likely)
if n=1000, $p=(1-.01)^{298}=.05$ (unlikely)

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.94

94



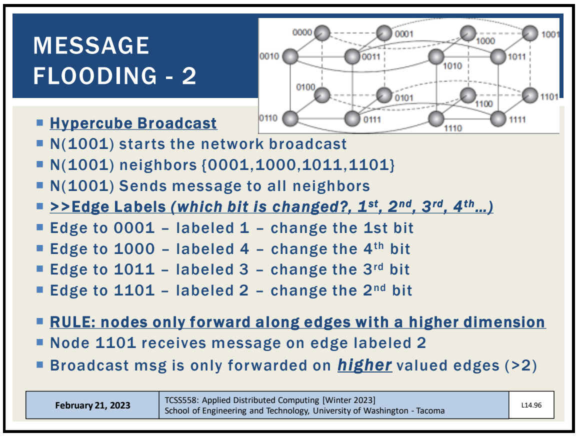
MESSAGE FLOODING

- For deterministic topologies (such as hypercube), design of efficient flooding scheme is much simpler
- If the overlay network is structured, this gives us a deterministic topology
- Schlosser et al [2002] – offer simple and efficient broadcasting scheme that relies on keeping track of neighbors per dimension

0000
0001
1000
1001
0010
0011
1010
1011
0100
0101
1100
1101
0110
0111
1110
1111

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.95

95



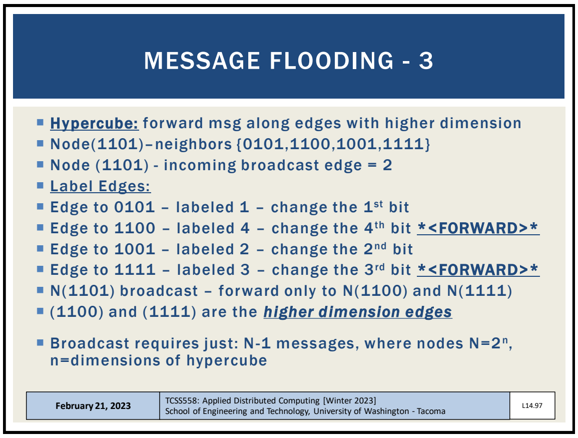
MESSAGE FLOODING - 2

0000
0001
1000
1001
0010
0011
1010
1011
0100
0101
1100
1101
0110
0111
1110
1111

- Hypercube Broadcast
- N(1001) starts the network broadcast
- N(1001) neighbors {0001,1000,1011,1101}
- N(1001) Sends message to all neighbors
- *>>Edge Labels (which bit is changed?, 1st, 2nd, 3rd, 4th…)*
- Edge to 0001 – labeled 1 – change the 1st bit
- Edge to 1000 – labeled 4 – change the 4th bit
- Edge to 1011 – labeled 3 – change the 3rd bit
- Edge to 1101 – labeled 2 – change the 2nd bit
- RULE: nodes only forward along edges with a higher dimension
- Node 1101 receives message on edge labeled 2
- Broadcast msg is only forwarded on *higher* valued edges (>2)

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.96

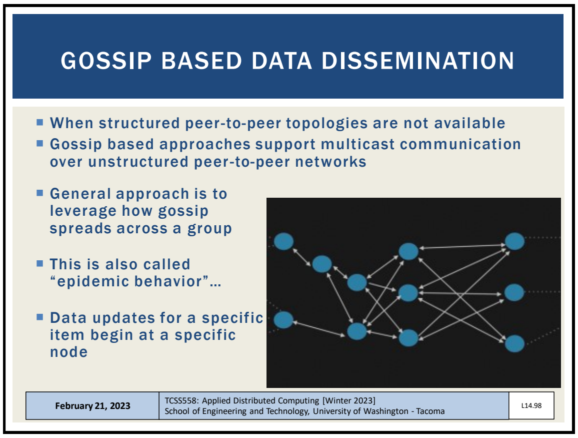
96



MESSAGE FLOODING - 3

- **Hypercube:** forward msg along edges with higher dimension
- Node(1101)–neighbors {0101,1100,1001,1111}
- Node (1101) - incoming broadcast edge = 2
- **Label Edges:**
- Edge to 0101 – labeled 1 – change the 1st bit
- Edge to 1100 – labeled 4 – change the 4th bit ***<FORWARD>***
- Edge to 1001 – labeled 2 – change the 2nd bit
- Edge to 1111 – labeled 3 – change the 3rd bit ***<FORWARD>***
- N(1101) broadcast – forward only to N(1100) and N(1111)
- (1100) and (1111) are the ***higher dimension edges***
- Broadcast requires just: N-1 messages, where nodes $N=2^n$, n=dimensions of hypercube

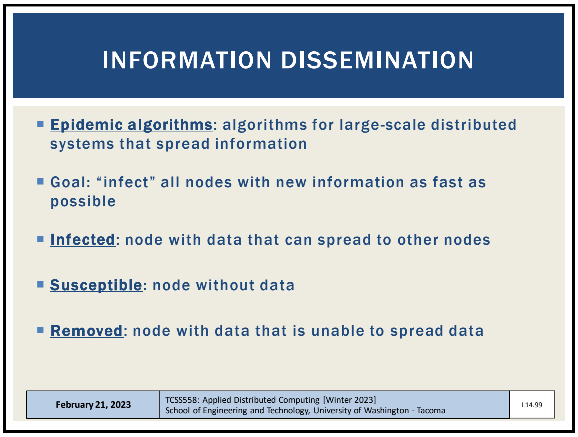
97



GOSSIP BASED DATA DISSEMINATION

- When structured peer-to-peer topologies are not available
- Gossip based approaches support multicast communication over unstructured peer-to-peer networks
- General approach is to leverage how gossip spreads across a group
- This is also called "epidemic behavior"…
- Data updates for a specific item begin at a specific node

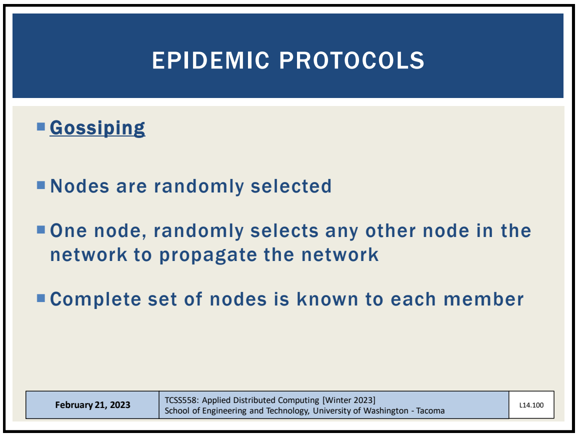
98



INFORMATION DISSEMINATION

- **Epidemic algorithms**: algorithms for large-scale distributed systems that spread information
- Goal: "infect" all nodes with new information as fast as possible
- **Infected**: node with data that can spread to other nodes
- **Susceptible**: node without data
- **Removed**: node with data that is unable to spread data

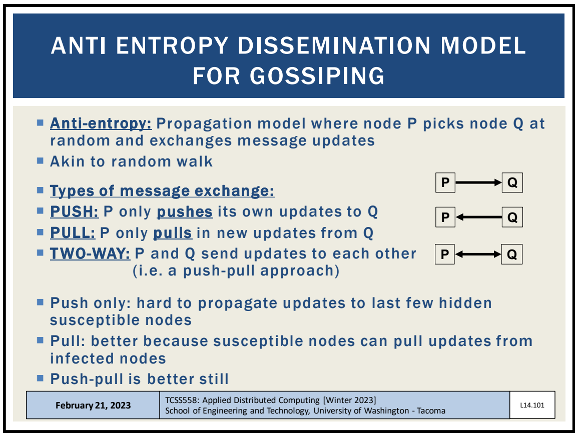
99



EPIDEMIC PROTOCOLS

- **Gossiping**
- Nodes are randomly selected
- One node, randomly selects any other node in the network to propagate the network
- Complete set of nodes is known to each member

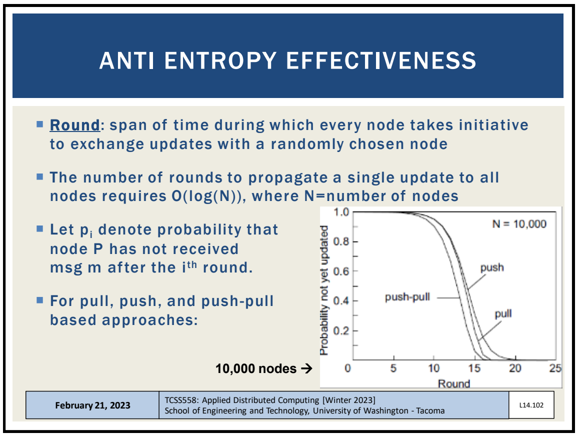
100



ANTI ENTROPY DISSEMINATION MODEL FOR GOSSIPING

- **Anti-entropy:** Propagation model where node P picks node Q at random and exchanges message updates
- Akin to random walk
- **Types of message exchange:**
- **PUSH:** P only **pushes** its own updates to Q
- **PULL:** P only **pulls** in new updates from Q
- **TWO-WAY:** P and Q send updates to each other (i.e. a push-pull approach)
- Push only: hard to propagate updates to last few hidden susceptible nodes
- Pull: better because susceptible nodes can pull updates from infected nodes
- Push-pull is better still

101

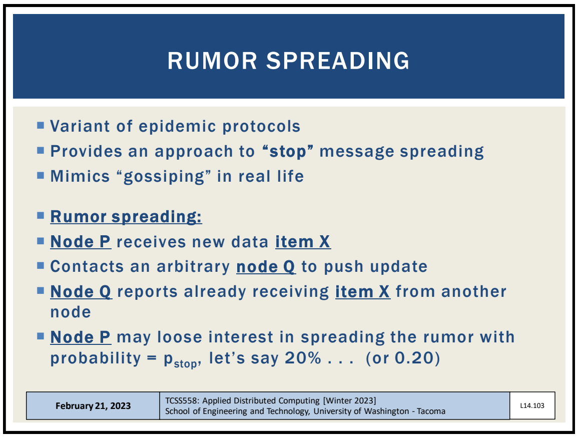


ANTI ENTROPY EFFECTIVENESS

- **Round**: span of time during which every node takes initiative to exchange updates with a randomly chosen node
- The number of rounds to propagate a single update to all nodes requires O(log(N)), where N=number of nodes
- Let p_i denote probability that node P has not received msg m after the i^{th} round.
- For pull, push, and push-pull based approaches:

10,000 nodes →

102

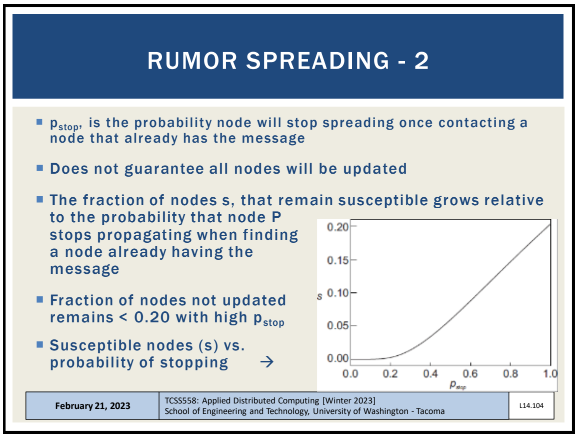


RUMOR SPREADING

- Variant of epidemic protocols
- Provides an approach to **"stop"** message spreading
- Mimics "gossiping" in real life
- **Rumor spreading:**
- **Node P** receives new data **item X**
- Contacts an arbitrary **node Q** to push update
- **Node Q** reports already receiving **item X** from another node
- **Node P** may loose interest in spreading the rumor with probability = p_{stop}, let's say 20% . . . (or 0.20)

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.103

103

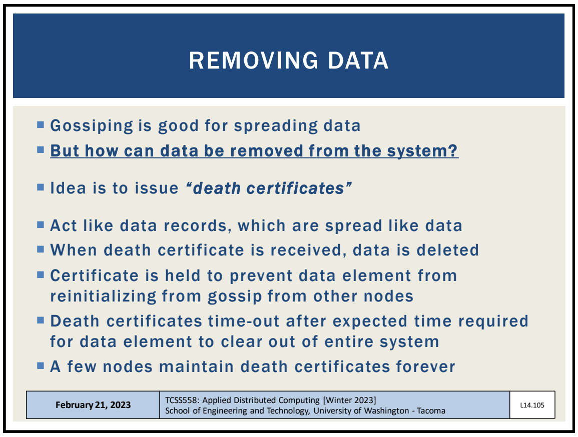


RUMOR SPREADING - 2

- p_{stop}, is the probability node will stop spreading once contacting a node that already has the message
- Does not guarantee all nodes will be updated
- The fraction of nodes s, that remain susceptible grows relative to the probability that node P stops propagating when finding a node already having the message
- Fraction of nodes not updated remains < 0.20 with high p_{stop}
- Susceptible nodes (s) vs. probability of stopping →

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.104

104

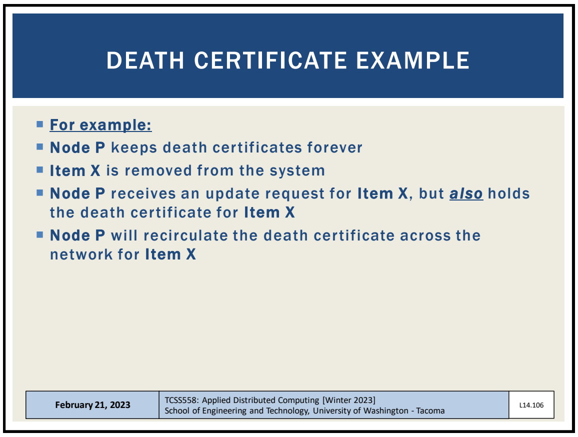


REMOVING DATA

- Gossiping is good for spreading data
- **But how can data be removed from the system?**
- Idea is to issue ***"death certificates"***
- Act like data records, which are spread like data
- When death certificate is received, data is deleted
- Certificate is held to prevent data element from reinitializing from gossip from other nodes
- Death certificates time-out after expected time required for data element to clear out of entire system
- A few nodes maintain death certificates forever

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.105

105

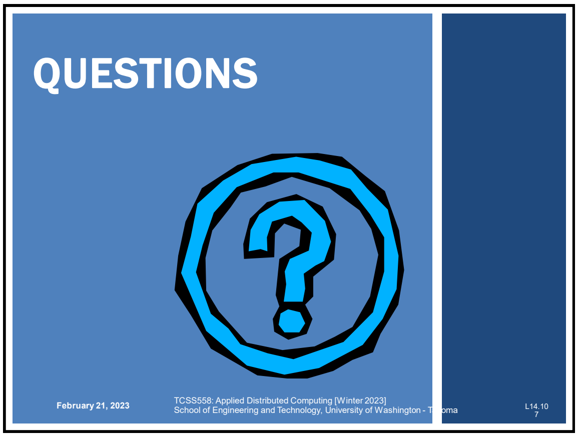


DEATH CERTIFICATE EXAMPLE

- **For example:**
- **Node P** keeps death certificates forever
- **Item X** is removed from the system
- **Node P** receives an update request for **Item X**, but ***also*** holds the death certificate for **Item X**
- **Node P** will recirculate the death certificate across the network for **Item X**

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - Tacoma | L14.106

106



QUESTIONS

February 21, 2023 | TCSS558: Applied Distributed Computing [Winter 2023] School of Engineering and Technology, University of Washington - T oma | L14.107

107