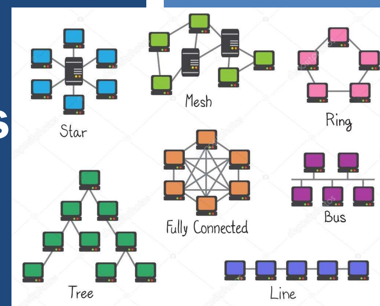
TCSS 558: APPLIED DISTRIBUTED COMPUTING



Chapter 4 - Processes

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

1

OBJECTIVES

- Assignment 1 – questions
- Feedback from 2/11
- Practice Midterm
- Chapter 4: Communication
- Chapter 4.1: Foundations
- Chapter 4.2: Remote Procedure Call
- Chapter 4.3: Message Oriented Communication

2

MIDTERM SCHEDULING SURVEY

- **TCSS 558B**
- Tuesday February 11 – 6 respondents (32%)
- Thursday February 13 – 7 respondents (37%) *(Internship fair @UW Seattle)*
- **Tuesday February 18 – 12 respondents (63%)** √
- No Preference – 2 respondents (11%)

- **Midterm Plan:**
- Content coverage - through 1st half of Lecture 11 on Feb 11th
- Practice midterm - 2nd half of Lecture 11 on Feb 11th
- February 13th – Will cover new material not on midterm
- Midterm Exam – Tuesday February 18th
- Exams returned no later than Tuesday February 25th

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.3

3

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (12 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- **Average – 6.16**

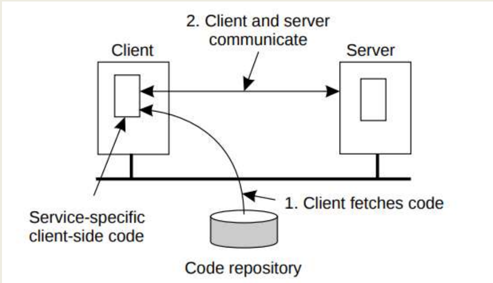
- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- **Average – 5.58**

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.4

4

FEEDBACK FROM 2/11

- Provide brief scenarios of code migration an organization may face
- Code migration in distributed systems:
- Receiver-initiated (pull)
  - Very common: use web browser to download recent version of client designed to interact with distributed system
  - Allows system to change
- Sender-initiated (push)
  - Often it is easier to move code than big data
    CODE SIZE – 15 MB
    DATA SIZE – 15 GB !!
  - Send data processing code (queries) to database



5

TYPES OF CODE MOBILITY

- Code on demand
  - User downloads application for local execution
  - E.g. recently updated client for distributed system, Javascript games
- Remote evaluation
  - Client uploads code to remote server
  - Remote server hosts BIG DATA
  - Client code queries/aggregates large remote datasets
  - Infeasible to run code locally
  - Network bandwidth insufficient to download data – time/cost prohibitive (e.g. cloud computing data egress charges)
- Mobile agents
  - Migration of code to different nodes in distributed system
  - Goal: improve load balancing, performance, fault tolerance

6

CH. 4 COMMUNICATION

L12.7

7

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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.8

8

CH. 4.1: FOUNDATIONS

L12.9

9

LAYERED PROTOCOLS

- Distributed systems lack shared memory
- All distributed system communication is based on sending and receiving low-level messages
  - P → Q
- Open Systems Interconnection Reference Model (OSI Model)
  - Open systems communicate with any other open system
  - Standards govern format, contents, meaning of messages
  - Formalization of rules forms a **communication protocol**

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.10

10

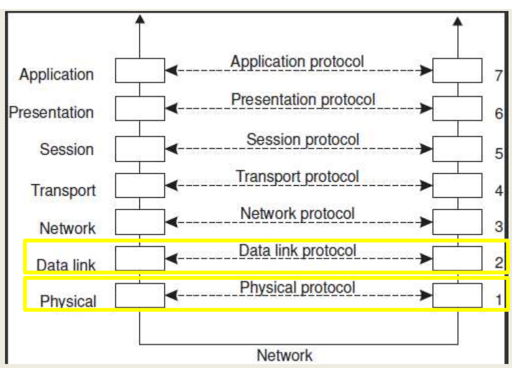
LAYERED PROTOCOLS - 2

- Protocols provide a <u>communication service</u>
- <u>Two service types:</u>
  - **<u>Connection-oriented</u>**: sender/receiver establish connection, negotiate parameters of the protocol, close connection when done
  - Physical example: telephone
  - **<u>Connectionless</u>**: No setup. Sender sends. Receiver receives.
  - Physical example: Mailing a letter

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.11

11

OSI MODEL REVISITED

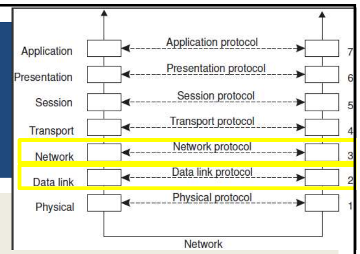


- Physical layer: just sends bits → … 0 0 0 1 0 1 1 0 1 1 …
- Data link layer: Groups bits into frames
  - Provides error correction via ***<u>checksum</u>***
  - Special bit pattern at start/end of frame

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.12

12

OSI MODEL - 2

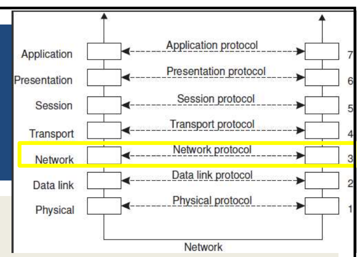


- Data link layer:
  - <u>**Checksum**</u>: computed by adding all bytes in frame in particular way
  - Added to message
  - Receiver removes checksum, recomputes checksum, and compares
  - If receiver and sender agree, frame is considered correct
  - Receiver can request failed frames to be resent
  - Frames assigned sequence numbers *<u>in the header</u>*
- Network layer:
  - Sometimes referred to as the *Internet layer*
  - On WANs sending msgs between client/server requires routing
  - Provides addressing using IPV4 (32-bit), IPV6 (64-bit)

February 13, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.13

13

OSI MODEL - 3

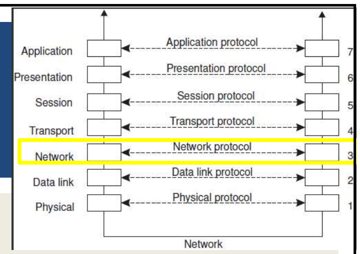


- Network layer:
  - Helps with routing network traffic
  - Shortest route (# of hops) may not be the best route
  - Minimizing delay (latency) is paramount
  - Routing algorithms: use long-term average network conditions, or try to adapt to changing conditions
  - ICMP Protocol: Internet Control Message Protocol
  - Not typically for sending data, used for diagnostic/control purposes
  - ICMP Examples: (*ping*, *traceroute*)

February 13, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.14

14

OSI MODEL - 4



- Internet Control Message Protocol (ICMP)
  - 8 bytes header: 4 fixed, 4 variable

ICMP Header Format

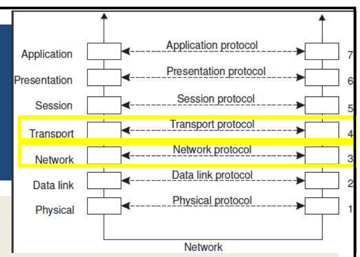
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Type								Code								Checksum															
4	32	Rest of Header																															

- Example message types:
- 0- echo reply (PING), 3- destination unreachable, 4- source quench (congestion control), 5- redirect message, 8- echo request (PING), 9- router advertisement
- Others: 10 (router solicitation), 11 (time exceeded), 12 (parameter problem), 13 (timestamp), 15 (info request), 16 (info reply), 17 (address mask request), 18 (address mask reply), 30-39 (traceroute), 40 (security failures), 42 (ext echo request)…255

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.15

15

OSI MODEL - 5



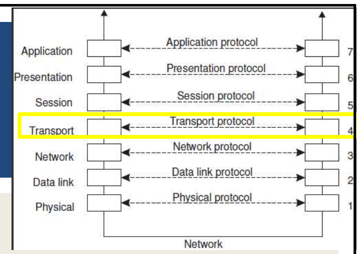
- Transport layer:
  - Provides reliable connections
  - Reorganizes packets arriving out of sequence
  - Requests delivery of missing packets

1. Breaks application layer protocol messages into pieces to transmit
2. Assigns messages sequence numbers
3. Sends all messages

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.16

16

OSI MODEL - 6

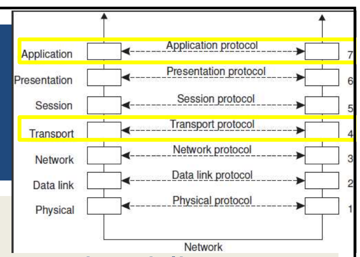


- Transport layer provides an infallible "message pipe"
  - Put messages in
  - Always come out undamaged, in correct order
- Transport layer protocols:
  - TCP: Transmission Control Protocol (connection-oriented)
  - UDP: Universal Datagram Protocol (connectionless)

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.17

17

OSI MODEL - 7

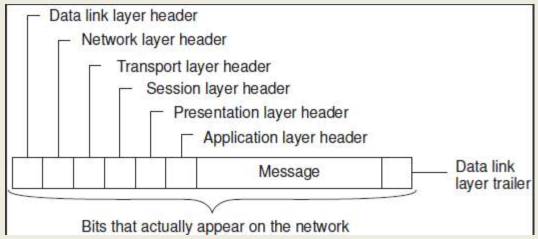


- Other transport protocols
  - Real-time transport protocol (RTP): real-time data, no data delivery guarantee
  - Streaming Control Transmission Protocol (SCTP): alternative to TCP
- Higher-level protocols:
- Session layer: mechanisms for opening, closing, managing session between communicating processes
- Presentation layer: deals with syntactical meaning of messages
  - Presentation services convert data among formats, for example:
  - from extended binary coded decimal interchange code (EBCDIC) to ASCII
- Application layer: protocols that don't fit into other layers
  - Many protocols: FTP, SFTP, HTTP, etc. etc.

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.18

18

OSI MODEL - 8



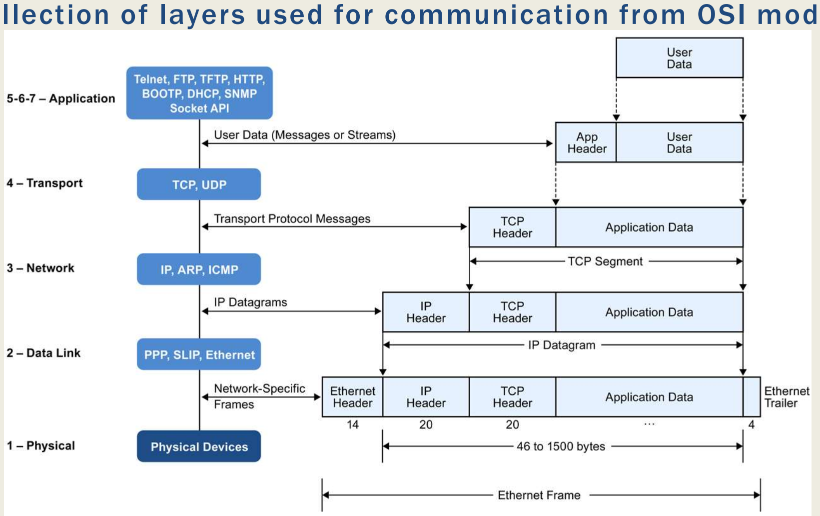
- Each OSI layer contributes overhead bits to the message
- Layers append data to front (and maybe end) of the message
- Receiver strips off headers as the message goes up the OSI model stack:

  physical → data-link → network → transport → application

19

PROTOCOL STACK

- Collection of layers used for communication from OSI model



20

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"

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.21

21

MIDDLEWARE PROTOCOLS - 2

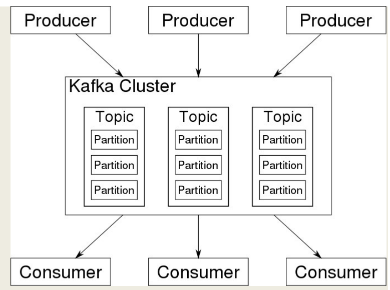
- Distributed commit protocols
  - Coordinate a group of processes (nodes)
  - Facilitate all nodes carrying out a particular operation
  - Or abort transaction
  - Provides distributed atomicity (all-or-nothing) operations
- Distributed locking protocols
  - Protect a resource from simultaneous access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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.22

22

MIDDLEWARE PROTOCOLS - 3

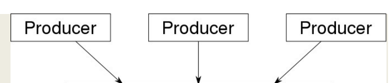
- **Message queueing services**
  - Support synchronization of data streams
  - Transfer real-time data
  - Distributed and scalable implementation



- **Multicast services**
  - Scale communication to thousands of receivers spread across the Internet

MIDDLEWARE PROTOCOLS - 3

- **Message queueing services**

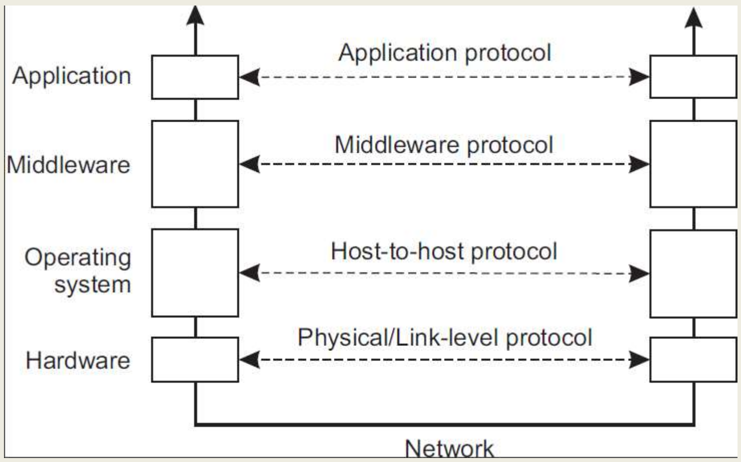


KEY: middleware protocols offer functionality to satisfy the software requirements of *many* applications

Middleware functions are general, application-independent in nature

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

ADAPTED REFERENCE MODEL



Combines network and transport

Physical and Data link

TYPES OF COMMUNICATION

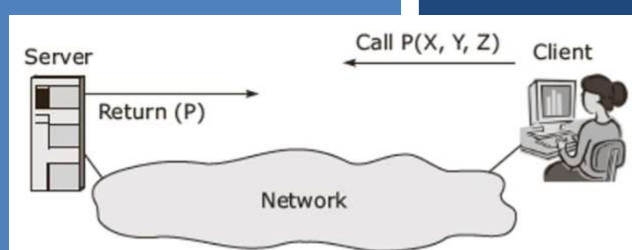
- Persistent communication
  - Message submitted for transmission is stored by communication middleware as long as it takes to deliver it
  - Example: email system (SMTP)
  - Receiver can be offline when message sent
  - Temporal decoupling (delayed message delivery)
- Transient communication
  - Message 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?**

TYPES OF COMMUNICATION - 2

- Asynchronous communication
  - Client does not block, continues doing other work
- Synchronous communication
  - Client blocks and waits
- Three types of blocking
  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
  - Block until message delivered to queue
- **Consider each type of blocking (1, 2, 3). Are these modes connectionless (UDP)? connection-oriented (TCP)?**

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.27

27



CH. 4.2: RPC

L12.28

28

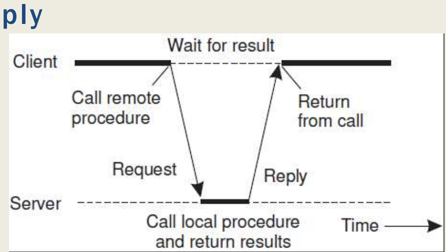
RPC – REMOTE PROCEDURE CALL

- 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)`

29

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***



30

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"

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.31

31

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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.32

32

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)

33

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*	

34

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**:

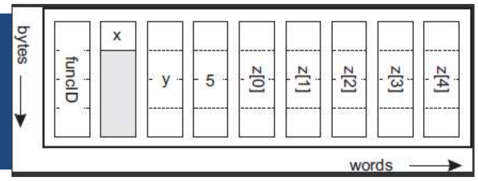
1. Forbid pointers altogether
2. Replace pass-by-reference with pass-by-value
   - Requires transferring entire object/array data over network
   - **Read-only optimization**: don't return data if unchanged on server
3. Passing global references
   - Example: file handle to file accessible by client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

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

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?

37

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

38

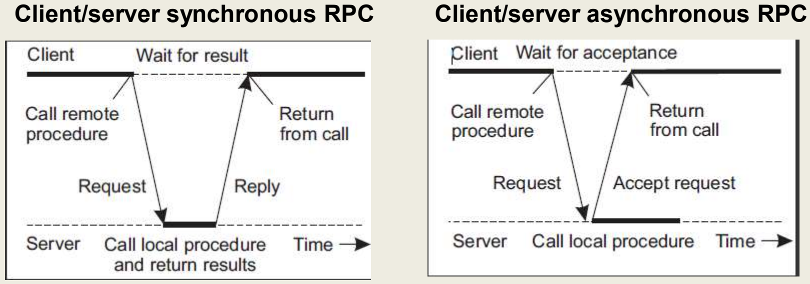
LANGUAGE BASED SUPPORT

- 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

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**



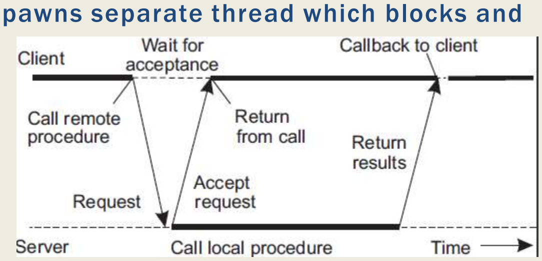
RPC VARIATIONS – 2

- 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…

41

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

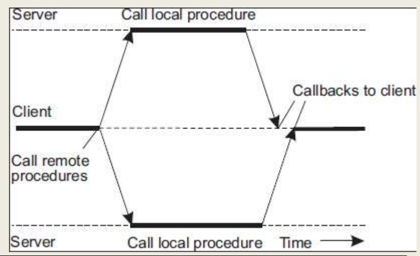


- **One-way RPCs**
  - Client **does not wait** for ***any*** server acknowledgement – it just goes…
- **Client polling**
  - Client (*using separate thread*) continually polls server for result

42

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



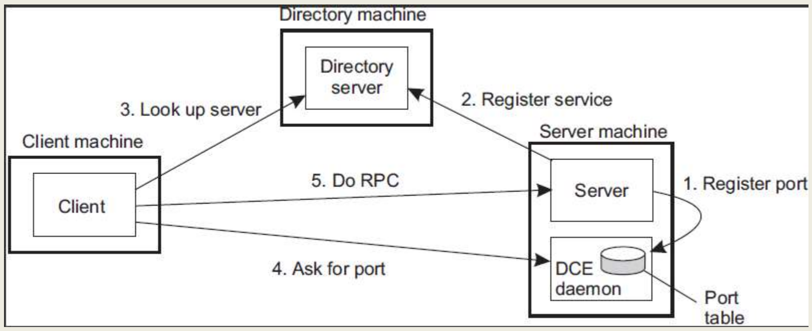
43

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

44

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

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

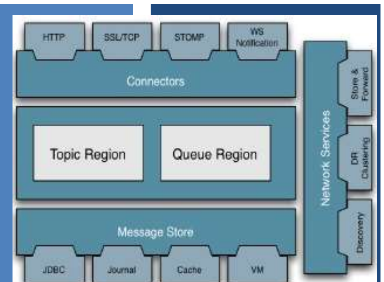
EXTRA: DCE – BINDING CLIENT TO SERVER

- 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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.47

47



Apache ActiveMQ

CH. 4.3: MESSAGE-ORIENTED COMMUNICATION

L12.48

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

February 13, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.49

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

February 13, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L12.50

50

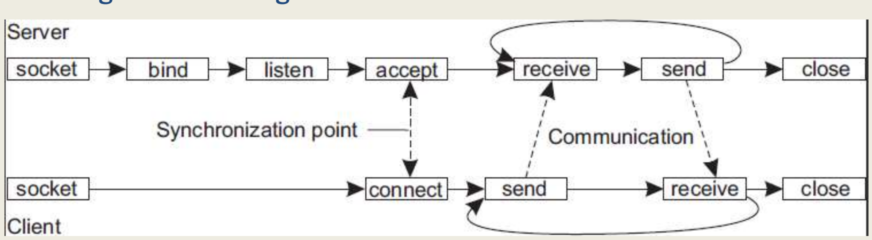
SERVER SOCKET OPERATIONS

- **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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.53

53

MESSAGE ORIENTED COMMUNICATION

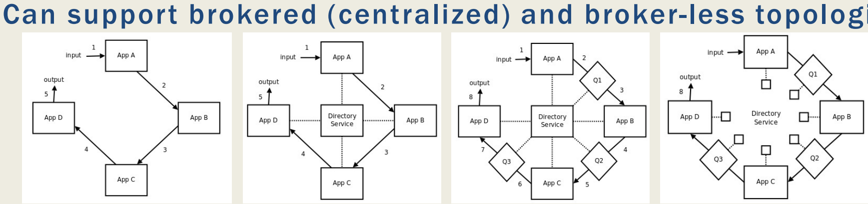
- RPC assumes that the ***client*** and ***server*** are running ***at the same time… (temporally coupled)***
- RPC communication is typically ***synchronous***
- When client and server are not running at the same time
- Or when communications should not be **blocked**…
- **This is a use case for message-oriented communication**
  - Synchronous vs. asynchronous
  - Messaging systems
  - Message-queueing systems

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.54

54

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



February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.55

55

ZEROMQ – 2

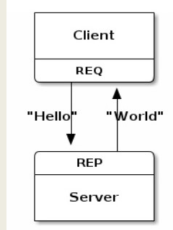
- ZeroMQ is TCP-connection-oriented communication
- Provides socket-like primitives with more functionality
  - Basic socket operations ***abstracted*** away
  - Supports many-to-one, one-to-one, and one-to-many connections
  - ***Multicast*** connections (one-to-many – single server socket simultaneously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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.56

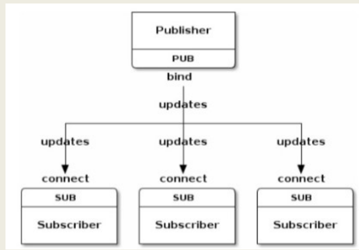
56

ZEROMQ - PATTERNS

- **Request-reply pattern**
  - Traditional client-server communication (e.g. RPC)
  - Client: request socket (REQ)
  - Server: reply socket (REP)



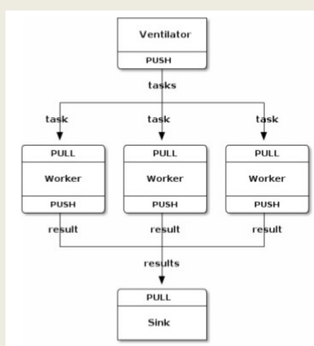
- **Publish-subscribe pattern**
  - Clients **subscribe** to messages **published** by servers
  - As in event-based coordination (Ch. 1)
  - Supports multicasting messages from server to multiple
  - Client: subscribe socket (SUB)
  - Server: publish socket (PUB)



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



58

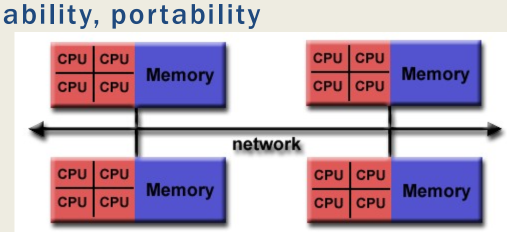
QUEUEING ALTERNATIVES

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

59

MESSAGE PASSING INTERFACE (MPI)

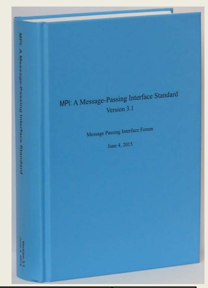
- MPI introduced – version 1.0 March 1994
- Message passing API for parallel programming: ***supercomputers***
- Communication protocol for parallel programming for: Supercomputers, High Performance Computing (HPC) clusters
- Point-to-point and collective communication
- Goals: high performance, scalability,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



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



February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.61

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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.62

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

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, <u>**do not block!**</u>

MESSAGE-ORIENTED-MIDDLEWARE

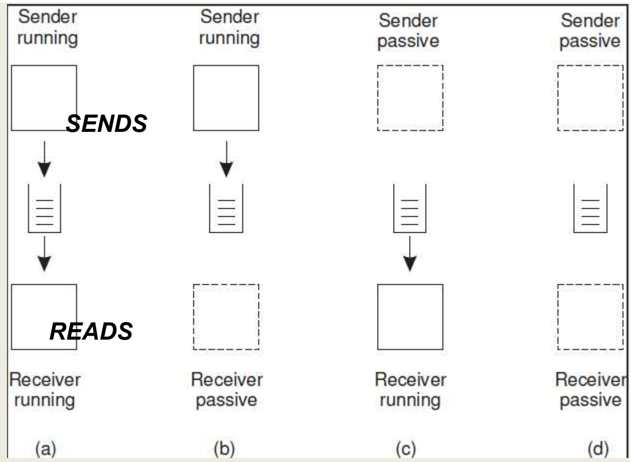
- Message-queueing systems
  - Provide extensive support for ***persistent*** asynchronous communication
  - In contrast to transient systems
  - Temporally decoupled: messages are eventually delivered to recipient queues
- Message transfers may take minutes vs. sec or ms
- Each application has its own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



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

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

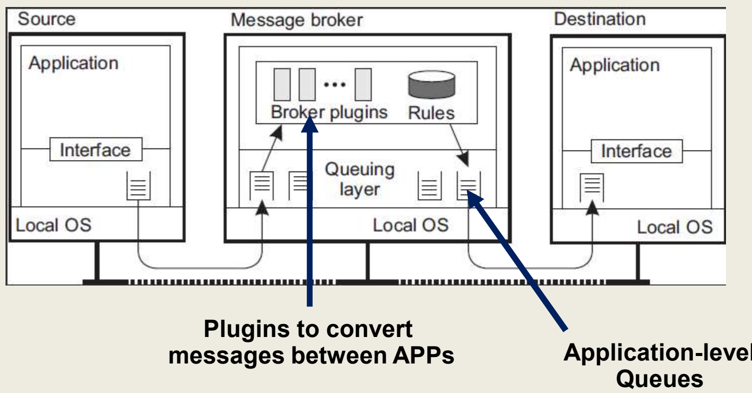
MESSAGE QUEUEING SYSTEMS ARCHITECTURE - 2

- **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



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

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.73

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)

February 13, 2020	TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma	L12.74

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

QUESTIONS

76