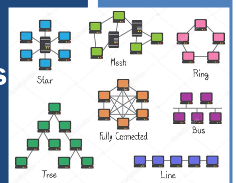
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Chapter 4 - Processes

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



1

OBJECTIVES

- Assignment 1 – questions
- Assignment 2 - questions
- Feedback from 2/20
- Chapter 4.2: Remote Procedure Call
- Chapter 4.3: Message Oriented Communication
- Chapter 4.4: Multicast Communication
- Chapter 6: Coordination

2

MATERIAL / PACE

- Please classify your perspective on material covered in today's class (9 respondents):
- 1-mostly review, 5-equal new/review, 10-mostly new
- Average – 5.88

- Please rate the pace of today's class:
- 1-slow, 5-just right, 10-fast
- Average – 5.44

3

FEEDBACK FROM 2/20

4

CH. 4 COMMUNICATION

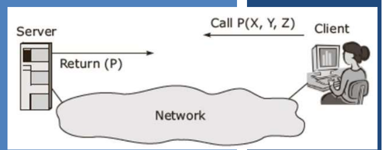
5

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"

6



CH. 4.2: RPC

L14.7

7

RPC – REMOTE PROCEDURE CALL

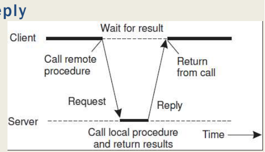
- In a nutshell,
- Allow programs to call procedures on other machines
- Process on **machine A** calls procedure on **machine B**
- Calling process on **machine A** is suspended
- Execution of the called procedure takes place on **machine B**
- Data transported from caller **(A)** to provider **(B)** 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)`

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.8

8

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***



February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.9

9

RPC - 3

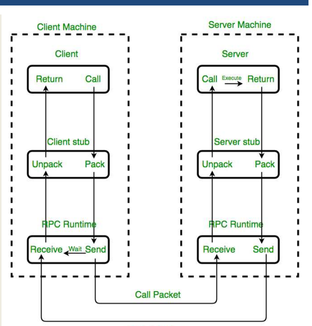
- Server packs procedure ***results*** and sends back to client.
- Client "***request***" call unblocks and data is unpacked
- Client can't tell method was called remotely over the network… ***except for network latency…***
- Call abstraction enables clients to invoke functions in alternate languages,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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.10

10

RPC STEPS

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



Implementation of RPC mechanism

11

RPC STEPS

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.12

12

PARAMETER PASSING

- **STUBS**: take parameters, pack into a message, send across network
- **EXAMPLE**: `newlist = append(data, dbList)`
- Parameters "data" and "dbList" must be sent over network
- Parameters are transferred as a series of bytes:

- Marshalling (serializing) → Unmarshalling (unserialize) of data
- Data is serialized into a "stream" of bytes

13

RPC: BYTE ORDERING

- Processor architectures may feature different byte ordering
- Older ARM CPUs: **Big-Endian**: write bytes left to right
- Intel: **Little-endian**: write bytes right to left
- Networks: typically transfer data in Big-Endian form
- Solution: transform data to machine/network independent format
- **RPC serialization:** 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*	

14

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

- (3) 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 pointer to file accessible by client and server via shared file system

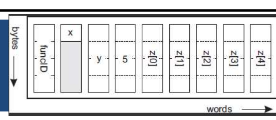
15

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
  - No stubs needed, can just share objects

16

STUB GENERATION



- `void func(char x; float y; int z[5])`
- 1-byte character transmits with 3-padded bytes
- 2-byte 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
- **RPC clients/servers:**
  - Need to agree on representation of simple data structures: int, char, floats w/ little endian
  - Must agree on protocol: TCP, UDP

17

STUB GENERATION - 2

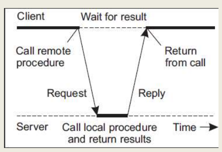
- Interfaces are specified using an Interface Definition Language (IDL)
- Interface specifications in IDL are used to generate language specific stubs
- IDL is compiled into client and server-side stubs
- Much of the plumbing for RPC involves maintaining boilerplate-code

18

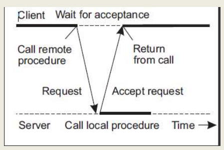
RPC SYNC VS. ASYNC

- 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



19

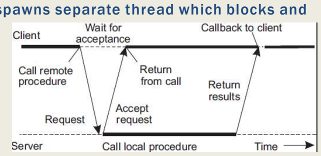
RPC SYNC VS. ASYNC– 2

- What would be a good use case for an asynchronous remote procedure call (RPC)?
- Use cases for asynchronous procedure calls:
  - Long running jobs allow client to perform alternate work in background ***without blocking…*** (in parallel)
  - Client may need to make multiple calls to multiple remote procedures at the same time…

20

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

21

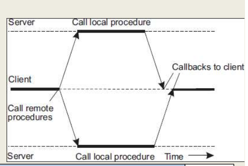
TYPES OF ASYNCHRONOUS RPC - 2

- **Rank the total volume of network traffic: (low, medium, high)**
- Deferred Synchronous RPC
- One-way RPC
- Client Polling
- **Which of these sustains an original TCP connection until the request is complete?**
- Deferred Synchronous RPC
- One-way RPC
- Client Polling

22

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 (i.e. race)*
  - Divide and conquer: multiple RPC calls work in parallel on different parts of dataset, client aggregates results



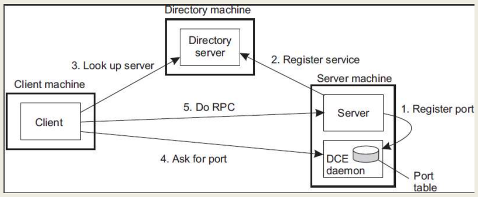
23

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

24

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.25

25

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.26

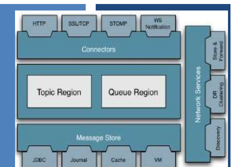
26

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.27

27



Apache ActiveMQ

CH. 4.3: MESSAGE-ORIENTED COMMUNICATION

L14.28

28

4.3 - MESSAGE ORIENTED COMMUNICATION

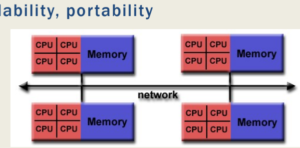
- **Topics:**
- Message passing interface (MPI)
- Message oriented middleware
  - Message queueing systems
  - Advanced message queueing protocol (AMQP)

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.29

29

MESSAGE PASSING INTERFACE (MPI)

- MPI introduced – version 1.0 March 1994
- Message passing API for parallel programming: ***supercomputers***
- **MPI is a communication protocol** for parallel programming on: Supercomputers, High Performance Computing (HPC) clusters
- Enables point-to-point and collective communication among nodes
- Goals: high performance, scalability, portability
- Most implementations in C, C++, Fortran
- **OpenMPI** – open source implementation for x86

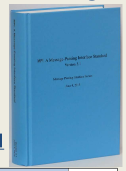


February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.30

30

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.31

31

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
  - *Similar to vendor-lock w/ cloud computing*
- Led to development of MPI
  - To support transient (non-persistent) communication for parallel programming

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.32

32

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.33

33

COMMON MPI FUNCTIONS

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

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.34

34

MESSAGE-ORIENTED-MIDDLEWARE (MOM)

- **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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.35

35

MESSAGE QUEUEING SYSTEMS: USE CASES

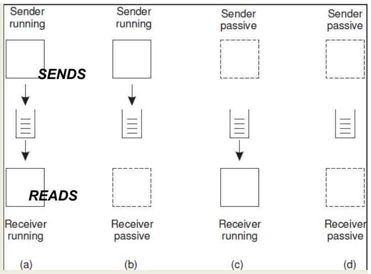
- Enables communication between applications, or sets of processes
  - User applications (service-oriented)
  - App-to-database
  - To support distributed real-time computations
- Use cases
  - Batch processing, Email, workflow, groupware, routing subqueries

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.36

36

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…



37

MESSAGE QUEUEING SYSTEMS - 2

- **Objective:** PROVIDE 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

38

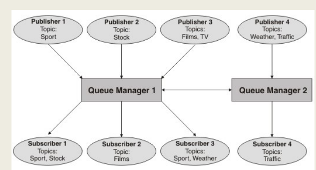
MESSAGE QUEUEING SYSTEMS - 3

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

39

MESSAGE QUEUEING SYSTEMS ARCHITECTURE

- **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



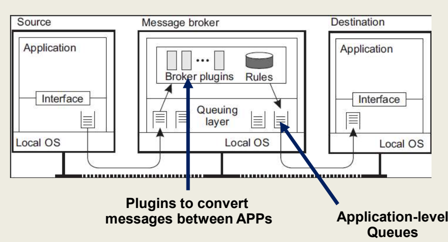
40

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

41

MESSAGE BROKER ORGANIZATION



42

ADVANCED MESSAGE QUEUEING PROTOCOL (AMQP)

- Message-queueing systems initially developed to enable legacy applications to interoperate
- Many are proprietary solutions, ***so not very open***
- e.g. Microsoft Message Queueing service, Windows NT 1997
- **Goal for common queueing protocols:** Decouple inter-application communication to "open" messaging-middleware
- **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+

43

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

44

AMQP MESSAGING

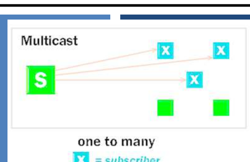
- AMQP nodes: producer, consumer, queue
- Producer/consumer: represent regular applications
- Queues: store/forward messages
- Persistent messaging:
- **Messages** can be marked ***durable***
- Durable messages can only be delivered by nodes able to recover in case of failure
- Non-failure resistant nodes must reject durable messages
- **Source/target** nodes can be marked ***durable***
- Track what is durable (node state, node+msgs)

45

MESSAGE-ORIENTED-MIDDLEWARE EXAMPLES:

- **Open source 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

46



Apache ActiveMQ

CH. 4.4: MULTICAST COMMUNICATION

47

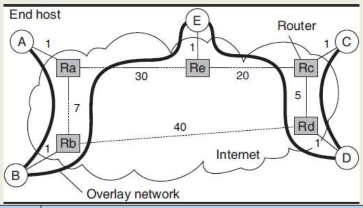
MULTICAST COMMUNICATION

- 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

48

NETWORK STRUCTURE

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



February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.49

49

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

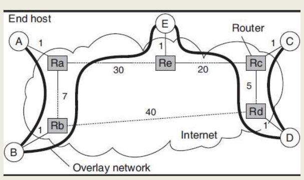
February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.50

50

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



February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.51

51

MULTICAST TREE METRICS - 2

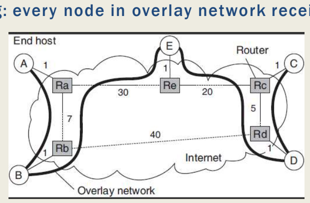
- **Stretch (Relative Delay Penalty RDP)**
- CONSIDER routing from B to C
- ***What is the Stretch?***
- Stretch (delay ratio) = Overlay-delay / Underlying-delay
- *Overlay:* B→Rb→Ra→Re→E→Re→Rc→Rd→D→Rd→Rc→ C = 73
- *Underlying:* B→Rb→Rd→Rc→C = 47
- Stretch = 73 / 47 = 1.55
- Captures additional time (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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.52

52

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?

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.53

53

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.54

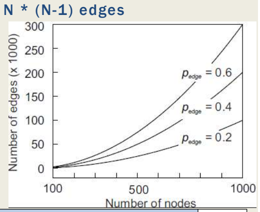
54

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.55

55

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.56

56

PROBABILISTIC FLOODING

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

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.57

57

PROBABILISTIC FLOODING

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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.58

58

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)

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.59

59

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.60

60

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.61

61

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.62

62

PROBABILISTIC FLOODING

- ….*Washington state in winter?*
- W ...ce 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 $p=(1-p_{flood})^n$

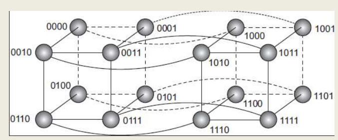
if n=10, $p=(1-.01)^{10}=.904$ (pretty likely)
if n=100, $p=(1-.01)^{100}=.366$ (less likely)
if n=1000, $p=(1-.01)^{298}=.05$ (unlikely)

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.63

63

MESSAGE FLOODING

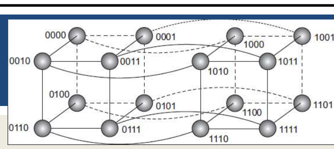
- For deterministic topologies (such as hypercube), design of efficient flooding scheme is much simpler
- If the overlay network is structured, this gives us a deterministic topology
- Schlosser et al [2002] – offer simple and efficient broadcasting scheme that relies on keeping track of neighbors per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



February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.64

64

MESSAGE FLOODING - 2



- **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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.65

65

MESSAGE FLOODING - 3

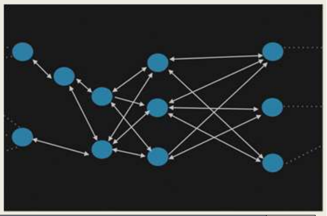
- **Hypercube:** forward msg along edges with higher dimension
- Node(1101)–neighbors {0101,1100,1001,1111}
- Node (1101) - incoming broadcast edge = 2
- **Label Edges:**
- Edge to 0101 – labeled 1 – change the 1st bit
- Edge to 1100 – labeled 4 – change the 4th bit ***<FORWARD>***
- Edge to 1001 – labeled 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

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.66

66

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



67

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

68

EPIDEMIC PROTOCOLS

- For gossiping, nodes are randomly selected
- One node, can randomly select any other node in the network
- Complete set of nodes is known to each member

69

ANTI ENTROPY DISSEMINATION MODEL

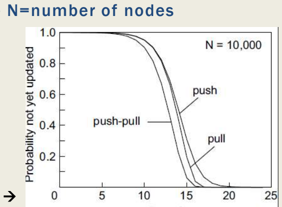
- **Anti-entropy:** Propagation model where node P picks node Q at random and exchanges message updates
- Akin to random walk
- **PUSH:** P only **pushes** its own updates to Q
- **PULL:** P only **pulls** in new updates from Q
- **TWO-WAY:** P and Q send updates to each other (i.e. a push-pull approach)
- 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

70

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 →



71

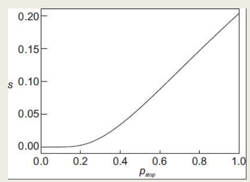
RUMOR SPREADING

- 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)

72

RUMOR SPREADING - 2

- 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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.73

73

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.74

74

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 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.75

75

CHAPTER 6 - COORDINATION

- 6.1 Clock Synchronization
  - Physical clocks
  - Clock synchronization algorithms
- 6.2 Logical clocks
  - Lamport clocks
  - Vector clocks
- 6.3 Mutual exclusion
- 6.4 Election algorithms
- 6.6 Distributed event matching *(light)*
- 6.7 Gossip-based coordination *(light)*

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.76

76

CHAPTER 6 - COORDINATION

- How can processes synchronize and coordinate data?
- Process synchronization
  - Coordinate cooperation to grant individual processes temporary access to shared resources (e.g. a file)
- Data synchronization
  - Ensure two sets of data are the same (data replication)
- Coordination
  - Goal is to manage interactions and dependencies between activities in the distributed system
  - Encapsulates synchronization

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.77

77

COORDINATION - 2

- Synchronization challenges begin with **time**:
  - How can we synchronize computers, so they all agree on the time?
  - How do we measure and coordinate when things happen?
- Fortunately, for synchronization in distributed systems, it is often sufficient to only agree on a relative ordering of events
  - E.g. not actual time

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.78

78

COORDINATION - 3

- Groups of processes often appoint a **coordinator**
- **Election algorithms** can help elect a leader
- Synchronizing access to a shared resource is achieved with **distributed mutual exclusion** algorithms
- Also in chapter 6:
  - Matching subscriptions to publications in publish-subscribe systems
  - Gossip-based coordinate problems:
    - Aggregation
    - Peer sampling
    - Overlay construction

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.79

79

QUESTIONS

February 25, 2020 | TCSS558: Applied Distributed Computing [Winter 2020] School of Engineering and Technology, University of Washington - Tacoma | L14.80

80