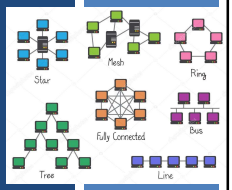
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Chapter 3 – Processes
Chapter 4 - Communication

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



OBJECTIVES

- Homework 0 – revisisted
- Homework 2
- Midterm Review

- Chapter 3 Processes
  - 3.5 Code Migration
- Chapter 4 Communication
  - 4.1

CH. 3.5: RESOURCE (CODE) MIGRATION

RESOURCE MIGRATION

- **Goal**: support on-the-fly reorganization of distributed systems
- At times there is interest in resource migration

- Can consider various types of resource migration
  - **Code migration**: source code, libraries
  - **Process migration**: a running job/task
  - **VM migration**: an entire virtual server!

CODE MIGRATION

- Distributed systems can support more than **passing data**
- Some situations call for **passing programs** (e.g. *code*)
- **Live migration** – moving code while it is executing
- **Portability** – transferring code (running or not) across heterogeneous systems:
  Mac OS X → Windows 10 → Linux
- Code migration enables ***flexibility*** of distributed systems
  - Topologies can be dynamically reconfigured on-the-fly

PROCESS MIGRATION

- Move an entire process from one node to another
- Motivation is always to address performance
- Process migration is slow, costly, and intricate
  - Need to pause, save intermediate state, move, resume
  - Consider application ***specific*** vs. ***agnostic*** approaches
- What would be:
  an **application agnostic** approach to migration?
  an **application specific** approach?
- What are advantages and disadvantages of each?

PROCESS MIGRATION - 2

- Move processes: from heavily loaded → lightly loaded nodes
- When do we consider a node as heavily loaded?
  - Load average
  - CPU utilization
  - CPU queue length
- Which process(es) should be moved?
  - Must consider ***resource requirements*** for the task
- Where should process(es) be moved to?

VM MIGRATION

- In some cases may look to migrate entire **Virtual Machine**
- **Motivations:**
  - Off-loading machines: reduce load on oversubscribed servers
  - Load machine: ensure machine has enough work to do
  - Idle servers: minimize total # of servers to save energy/cost
  - Maintenance: power down servers for HW repair/upgrade
- **VM migration:**
- Migrate complete VMs with apps to lightly loaded hosts
- Generally, VM migration is easier than process migration
- **Which is faster process or VM migration?**
- **Is VM migration application specific or agnostic?**

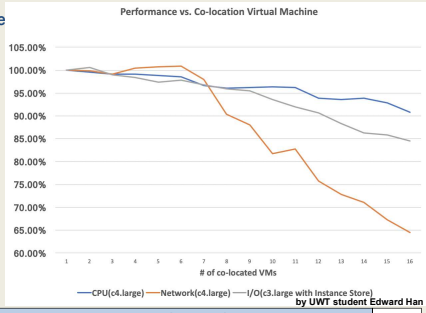
REASONS TO MIGRATE: VM OVERPROVISIONING RESOURCE CONTENTION FROM CO-LOCATED VMS

Run performance benchmarks in parallel

c4.large
CPU (y-cruncher)
Network (iPerf)

c3.large
Disk (pgbench)

1 to 16 co-resident VMs



by UWT student Edward Han

LINUX CRIU TOOL

- Linux (CRIU) Checkpoint restore in userspace
- Linux tool: **https://www.criu.org/**
- Supports freezing a running application (or part of it) to create a checkpoint to persistent storage (e.g. disk) as a collection of files.
  - This means saving the state of RAM to disk
- Can use checkpoint files to restore and run the application from the point it was frozen at.
- Distinctive feature of CRIU is that it can be run in the ***user space*** (CPU user mode), rather than in kernel mode.
- CRIU can save a Docker container's state for migration elsewhere

LOAD DISTRIBUTION ALGORITHMS

- Make decisions concerning allocation and redistribution of tasks across machines
- Provide resource management for compute intensive systems
- Often CPU centric
  - Algorithms should also account for other resources
  - Network capacity may be larger bottleneck that CPU capacity

WHEN TO MIGRATE?

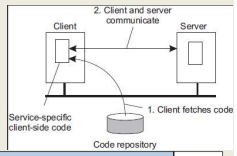
- Decisions to migrate code often based on qualitative reasoning or adhoc decisions vs. formal mathematical models
  - Difficult to formalize solutions due to heterogeneous composition and state of systems and networks
- **Is it better to migrate code or data?**
- **What factors should be considered?**
  - Size of code
  - Size of data
  - Available network transfer speed
  - Cost of data transfer
  - Processing power of nodes
  - Cost of processing
  - Are there security requirements for the data?

APPROACHES TO CODE MIGRATION

- Traditional clients
  - Client interacts with server using specific protocol
  - Tight coupling of client->server limits system flexibility
  - Difficult to change protocol when there are ***many*** clients
- Dynamic web clients
  - Web browser downloads client code immediately before use
  - New versions can readily be distributed

DYNAMIC WEB CLIENTS

- Advantages
  - Client code loaded in as necessary
  - Discarded when no longer needed
  - Can easily change the client/server protocol
- Disadvantages
  - Security: we have to trust the code
  - Downloading client requires network bandwidth & time



CODE MIGRATION

- Sender-initiated: (upload the code)… e.g. Github
- Receiver-initiated: (download the code)… e.g. web broswer
- **Remote cloning**
  - Produce a copy of the process on another machine while parent runs

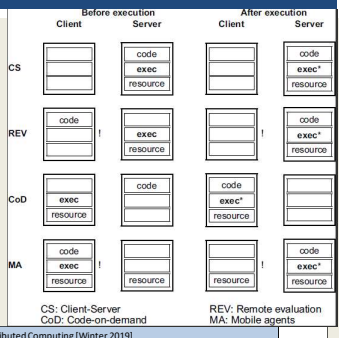
CODE MIGRATION - 2

- What program segments are migrated?
  - ***Code*** segment
  - ***Resource*** segment (device info)
  - ***Execution*** segment (process info: data, state, stack, PC)
- **Weak mobility**
  - Only ***code*** segment, no state
  - Code always restarts
- **Strong mobility**
  - ***Code*** + ***execution*** segment
  - Process stopped, state saved, moved, resumed
  - Represents true ***process migration***

CODE MOBILITY TYPES

- CS: Client-Server
- REV: Remote Evaluation
- CoD: Code-on-demand
- MA: Mobile agents
- Where does state get modified?
- State is stored in **exec**

** shows what is modified*



MIGRATION OF HETEROGENEOUS SYSTEMS

- **Hopeful assumption**: code will always work at new node
- Invalid if node architecture is different (*heterogeneous*)
- What approaches are available to migrate code across heterogeneous systems?
- Intermediate code
  - 1970s Pascal: generate machine-independent intermediate code
  - Programs could then run anywhere
  - **Today:** web languages: Javascript, Java
- VM Migration

VIRTUAL MACHINE MIGRATION

- Four approaches to transfer:

1. **PRECOPY**: Push all memory pages to new machine *(slow)*, resend modified pages later, transfer control
2. **STOP-AND-COPY**: Stop the VM, migrate memory pages, start new VM
3. **ON DEMAND**: Start new VM, copy memory as needed
4. **HYBRID**: PRECOPY followed by brief STOP-AND-COPY

- **What are some advantages and disadvantages of 1-4?**
  - See next slide

1. **PRECOPY**: Push all memory pages to new machine *(slow)*, resend modified pages later, transfer control
2. **STOP-AND-COPY**: Stop the VM, migrate memory pages, start new VM
3. **ON DEMAND**: Start new VM, copy memory pages as needed
4. **HYBRID**: PRECOPY and followed by brief STOP-AND-COPY

- **What are some advantages and disadvantages of 1-4?**
  - 1/3: no loss of service
  - 4: fast transfer, minimal loss of service
  - 2: fastest data transfer
  - 3: new VM immediately available
  - 1: must track modified pages during full page copy
  - 2: longest downtime - unacceptable for live services
  - 3: prolonged, slow, migration
  - 3: original VM must stay online for quite a while
  - 1/3: network load while original VM still in service

CH. 4 COMMUNICATION

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

Content consists of review and additional building on Ch 2/3

CH. 4.1: FOUNDATIONS

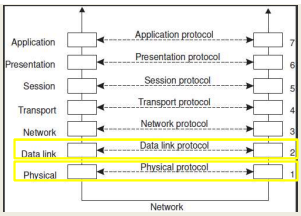
LAYERED PROTOCOLS

- Distributed systems lack shared memory
- All 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**

LAYERED PROTOCOLS - 2

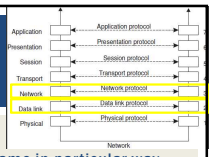
- Protocols provide a **communication service**
  - **Connection-oriented**: sender/receiver establish connection, negotiate parameters of the protocol, close connection when done
  - Physical example: telephone
  - **Connectionless services**: No setup. Sender sends. Receiver receives.
  - Physical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

OSI MODEL REVISITED



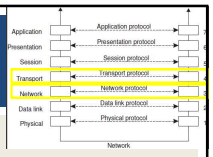
- Physical layer: just sends bits
- Data link layer: Groups bits into frames
  - Provides error correction via ***checksum***
  - Special bit pattern at start/end of frame

OSI MODEL - 2



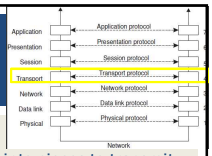
- Data link layer:
  - Checksum: computed by adding all bytes in frame in particular way
  - Added to message
  - Receiver removes checksum, recomputes checksum, and compares
  - If receiver and sender agree, frame is considered correct
  - Receiver can request failed frames to be resent
  - Frames assigned sequence numbers *in the header*
- 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)

OSI MODEL - 3



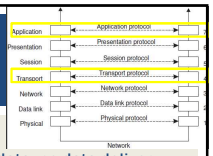
- Network layer:
  - Helps with routing network traffic
  - Shortest route (# of hops) may not be the best route
  - Minimizing delay (latency) is paramount
  - Routing algorithms: use long-term average network conditions, or try to adapt toe changing conditions
  - ICMP Protocol: Internet Control Message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

OSI MODEL - 4



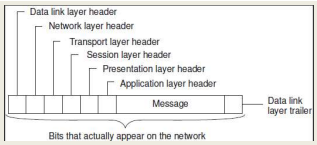
- Transport layer:
  1. Breaks application layer protocol messages into pieces to transmit
  2. Assigns messages sequence numbers
  3. Sends all messages
- Transport layer provides an infallible "message pipe"
  - Put messages in
  - Always come out undamaged, in correct order
- Transport layer protocols:
  - TCP: Transmission Control Protocol (connection-oriented)
  - UDP: Universal Datagram Protocol (connectionless)

OSI MODEL - 5



- 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: rarely used
  - Presentation layer: meaning of the bits;
  - Application layer: protocols that don't fit into other layers
    - Many protocols: FTP, SFTP, HTTP, etc. etc.

OSI MODEL - 2

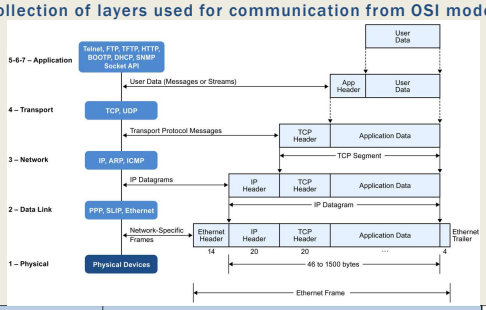


- OSI layers contribute overhead bits to the message
- Layers append data to front (and maybe end) of the message
- Receiving end strips off layers as the message goes up the OSI model stack:

  physical → data-link → network → transport → application

PROTOCOL STACK

- Collection of layers used for communication from OSI model



MIDDLEWARE PROTOCOLS

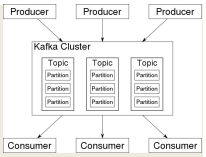
- Communication frameworks/libraries
- Reused by multiple applications
- Provided needed functions apps build and depend on
- Example:
  - **Authentication protocols**: supports granting users and processes access to authorized resources
  - General, application-independent in nature
  - Doesn't fit as an "application specific" protocol
  - Considered as a "Middleware protocol"

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
  - Distributed objects

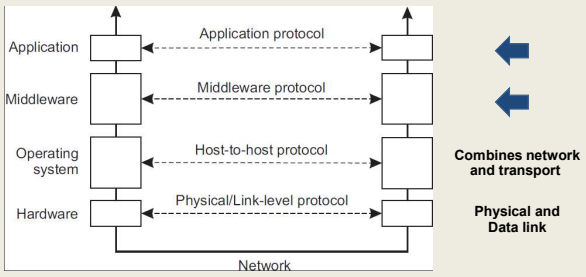
MIDDLEWARE PROTOCOLS - 3

- 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

ADAPTED REFERENCE MODEL

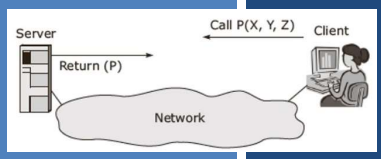


TYPES OF COMMUNICATION

- 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 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 synchronize until request has been delivered
  3. Sender waits until request is processed and result is returned
- Persistence + synchronization
  - Common scheme for message-queueing systems
- **Consider each type of blocking (1, 2, 3). Are these modes connectionless (UDP)? connection-oriented (TCP)?**



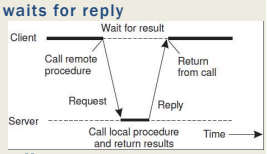
CH. 4.2: RPC

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 to server. Calls blocking receive routine and waits for reply
- **Server stub**: transforms incoming request into local procedure call
- Server blocks waiting for msg
- Server stub unpacks msg, calls server procedure
- ***It's as if the routine were called locally***



RPC - 3

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

RPC STEPS

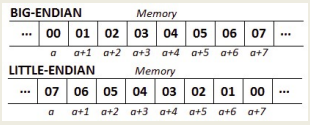
1. Client procedure calls client stub
2. Client stub builds message and calls OS
3. Client's OS send message to remote OS
4. Server OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server performs work, returns results to server-side stub
7. Server stub packs results 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 under stand how to unmarshal (unserialize) data
- Processor architecture 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



RPC: PASS-BY-REFERENCE

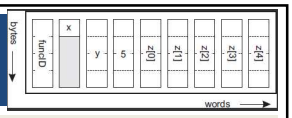
- Passing by value is straightforward
- Passing by reference is challenging
- Pointers only make sense on local machine owning the data
- Memory space of client and server are different
- Solutions to **RPC pass-by-reference**:

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 calling into the programming language
  - E.g. Java RMI

STUB GENERATION



- `void func(char x; float y; int z[5])`
- Character transmits with 3-padded bytes
- Float 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?

STUB GENERATION - 2

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

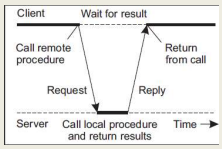
LANGUAGE BASED SUPPORT

- 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

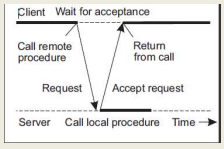
RPC VARIATIONS

- RPC: typically client 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

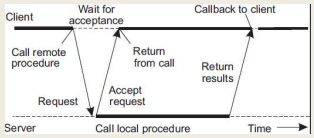


RPC VARIATIONS – 2

- 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
  - Client may need to make multiple service calls to multiple server backends at the same time…

TYPES OF ASYNCHRONOUS RPC

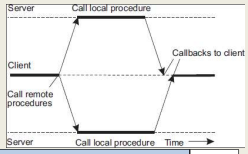
- **Deferred synchronous RPC**
  - Server performs ***CALLBACK*** to client
  - Client, upon making call, spawns separate thread which blocks and waits for call



- **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

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



RPC EXAMPLE: DISTRIBUTED COMPUTING ENVIRONMENT (DCE)

- **DCE**: basis for Microsoft's distributed computing object model (DCOM)
- Used in Samba – share windows filesystem via RPC
- Midleware system – provides layer of abstraction between OS and distributed applications
- Design for Unix, ported to all major operating systems
- Install DCE middleware on set of heterogeneous machines – distributed applications can then run and leverage resources
- Uses client/server model
- All communication via RPC
- DCE provides a daemon to track participating machines, ports

DCE – CLIENT/SERVER DEVELOPMENT

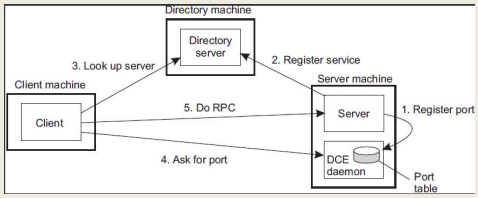
1. Create Interface definition language (IDL) files
   - IDL files contain Globally unique identifier (GUID)
   - GUIDs must match: client and server compare GUIDs to verify proper versions of the distributed object
   - 128-bit binary number
2. Next, add names of remote procs and params to IDL
3. Then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

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:

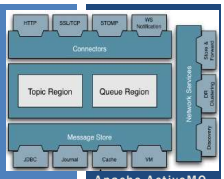
1. Server asks OS for a port, registers port with DCE daemon
2. Also, server registers with directory server, separate server that tracks DCE servers

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

CH. 4.3: MESSAGE-ORIENTED COMMUNICATION



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

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

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*)

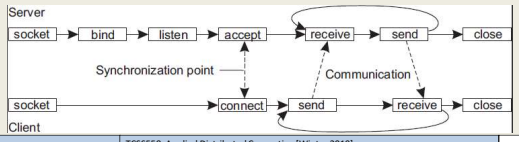
Operation	Description
socket	Create a new communication end point
bind	Attach local address to socket (IP / port)
listen	Tell OS what max # of pending connection requests should be
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
send	Send some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

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

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

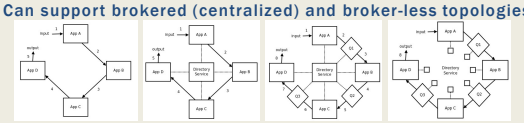


SOCKET COMMUNICATION

- 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

ZEROMQ – SOCKET LIBRARY

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

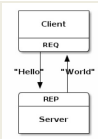


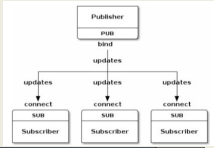
ZEROMQ – 2

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

ZEROMQ - PATTERNS

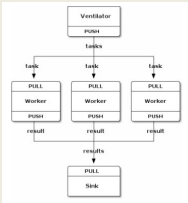
- **Request-reply pattern**
  - Traditional client-server communication (e.g. RPC)
  - Client: request socket (**REQ**)
  - Server: reply socket (**REP**)
- **Publish-subscribe pattern**
  - Clients **subscribe** to messages **published** by servers
  - As in event-based coordination (Ch. 1)
  - Supports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





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

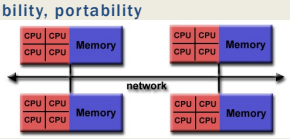


QUEUEING ALTERNATIVES

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

MESSAGE PASSING INTERFACE (MPI)

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



MOTIVATIONS FOR MPI

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

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

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	

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!**

MESSAGE-ORIENTED-MIDDLEWARE

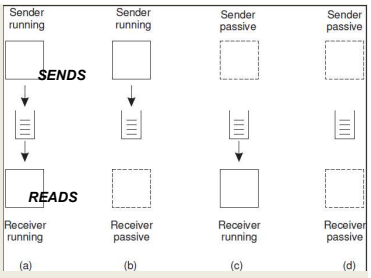
- **Message-queueing systems**
  - Provide extensive support for ***persistent*** asynchronous communication
  - In contrast to transient systems
  - Temporally decoupled: messages are eventually delivered to recipient queues
- Message transfers may take minutes vs. sec or ms
- Each application has its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

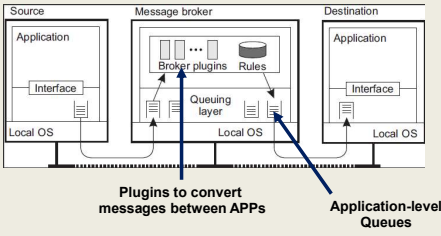
MESSAGE QUEUEING SYSTEMS ARCHITECTURE

- **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

MESSAGE QUEUEING SYSTEMS ARCHITECTURE - 2

- **ISSUES**:
- How do we route traffic between queue managers?
  - How are name-to-address mappings efficiently kept?
  - Each queue manager should be known to all others
- **Message brokers**
- Handle message conversion among different users/formats
- Addresses cases when senders and receivers don't speak the same protocol (language)
- Need arises for message protocol converters
  - "Reformatter" of messages
- Act as application-level gateway

MESSAGE BROKER ORGANIZATION



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+

AMQP - 2

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

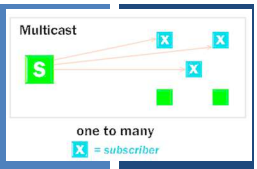
AMQP MESSAGING

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

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

MESSAGE-ORIENTED-MIDDLEWARE EXAMPLES:

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



CH. 4.4: MULTICAST COMMUNICATION

Apache ActiveMQ

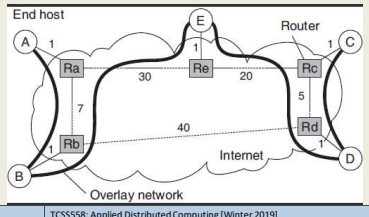
MULTICAST COMMUNICATION

- 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 invention

- 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

NETWORK STRUCTURE

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

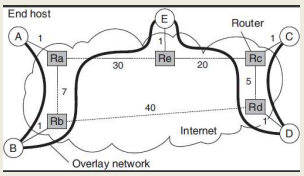


APPLICATION LEVEL TREE-BASED MULTICASTING

- 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

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



MULTICAST TREE METRICS - 2

- **Stretch (Relative Delay Penalty RDP)** for B to C routes:
- *Overlay:* B→Rb→Ra→Re→E→Re→Rc→Rd→D→Rd→Rc→ C = 73
- *Underlying:* B→Rb→Rd→Rc→C = 47
- 73 / 47 = 1.55

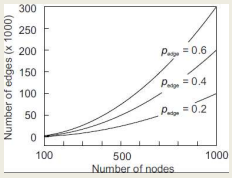
- **Tree cost:** Overall cost of the overlay network
- Ideally would like to minimize network costs
- Find a minimal spanning tree which minimizes total time for disseminating information

FLOOD-BASED MULTICASTING

- Broadcasting: every node in overlay receives message
- Key design issue: minimize the use of intermediate nodes for which the message is not intended
- Tree: if only the leaf nodes are to receive the multicast message, many intermediate nodes are involved
- Solution: construct an overlay network for each multicast group
- **Flooding**: each node simply forwards a message to each of its neighbors, except to the message originator

RANDOM GRAPHS

- When no information on the structure of the overlay network
- Assume network can be represented as a **Random graph**
- Probability Pedge that two nodes are joined
- Overlay will have: ½ * Pedge * N * (N-1) edges

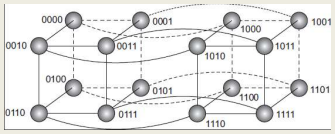


PROBABILISTIC FLOODING

- ***….Washington state in winter?***
- When a node is flooding a message, concept is to enforce a probability of message spread (p_{flood})
- Throttles message flooding based on a probability
- Implementation needs to consider # 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

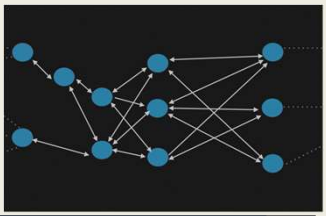
MESSAGE FLOODING

- 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
- **Hypercube:** nodes forward only to higher dimension nodes
- N(1001) broadcast will only go to N(1011) and N(1000)
- Broadcast requires just: N-1 messages, where nodes N=2^n, n=dimensions of hypercube



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



INFORMATION DISSEMINATION

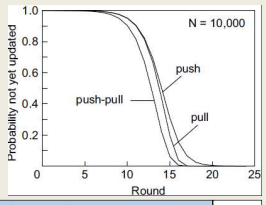
- **Epidemic algorithms**: algorithms for large-scale distributed systems that spread information
- Goal: "infect" all nodes with new information as fast as possible
- **Infected**: node with data that can spread to other nodes
- **Susceptible**: node without data
- **Removed**: node with data that is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

ANTI ENTROPY DISSEMINATION MODEL

- **Anti-entropy:** Propagation model where node P picks node Q at random and exchanges messages updates
- Akin to random walk
- **PULL:** P only **pulls** in new updates from Q
- **PUSH:** P only **pushes** its own updates to 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

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:

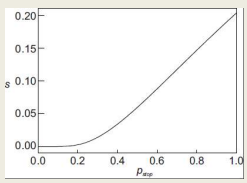


RUMOR SPREADING

- 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)

RUMOR SPREADING - 2

- Does not guarantee all nodes will be updated
- The fraction of nodes s, that remain susceptible is 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 →



DIRECTIONAL GOSSIPING

- Taking network topology into account can help
- When gossiping, nodes connected to only a few other nodes are more likely to be contacted
- **Epidemic protocols assume:**
- 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

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

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**

QUESTIONS