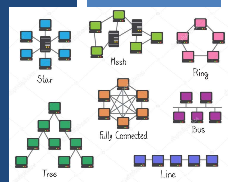
TCSS 558: APPLIED DISTRIBUTED COMPUTING

Chapter 3 – Processes
Chapter 4 - Communication

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



OBJECTIVES

- Homework 2
- Chapter 3 Processes
  - 3.5 Code Migration
- Chapter 4 Communication
- 4.1 Foundations- Protocols
- 4.2 Remote procedure call *(skip)*
- 4.3 Message-oriented communication
- 4.4 Multicast communication

FEEDBACK – 2/25

- **How easy is it to migrate a process – is data leak an issue to that needs to be accounted for? Or are there systems in place that prevent malicious parties from looking at data within a process?**
- If implementing a process migration protocol, which would be better:
  - UDP communication?
  - TCP communication?
- By adding SSH encryption to the transfer protocol (e.g. https) process data could be encrypted

FEEDBACK - 2

- **What if network is down during VM or process migration?**
- **Will data continue copying after the network connection is restored?**

CH. 3.5: RESOURCE (CODE) 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

L12.7

CH. 4 COMMUNICATION

L12.8

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

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.9

CH. 4.1: FOUNDATIONS

L12.10

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**

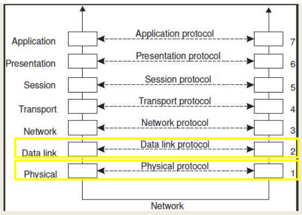
February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.11

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

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.12

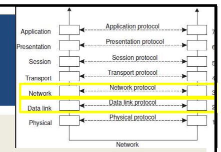
OSI MODEL REVISITED



- 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

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.13

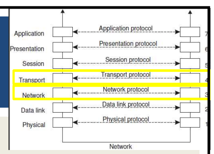
OSI MODEL - 2



- 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)

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.14

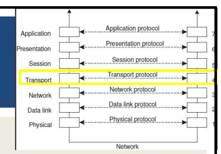
OSI MODEL - 3



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

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.15

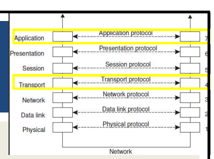
OSI MODEL - 4



- 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)

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.16

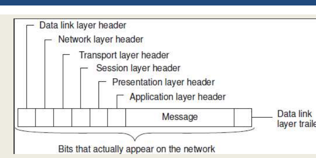
OSI MODEL - 5



- Other transport protocols
  - Real-time transport protocol (RTP): real-time data, no data delivery guarantee
  - Streaming Control Transmission Protocol (SCTP): alternative to TCP
- Higher layers
  - Session layer: rarely used
  - Presentation layer: meaning of the bits;
  - Application layer: protocols that don't fit into other layers
    - Most protocols: FTP, SFTP, HTTP, etc. etc.
    - Application protocols

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.17

OSI MODEL - 6



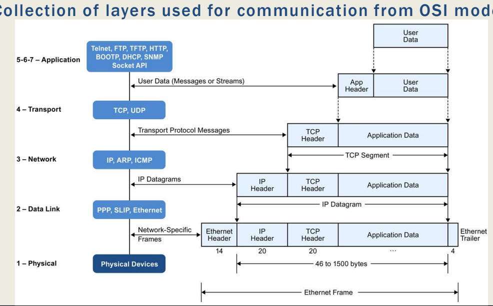
- OSI layers contribute overhead bits to the message
- Layers append data to front (and maybe end) of the message
- Receiving end strips off layers as the message goes up the OSI model stack:

  physical → data-link → network → transport → application

February 27, 2019 | TCSS558: Applied Distributed Computing [Winter 2019] School of Engineering and Technology, University of Washington - Tacoma | L12.18

PROTOCOL STACK

- Collection of layers used for communication from OSI model



MIDDLEWARE PROTOCOLS

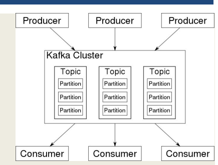
- Communication frameworks/libraries
- Reused by multiple applications
- Provided needed functions apps build and depend on
- Example:
  - **Authentication protocols**: supports granting users and processes access to authorized resources
  - General, application-independent in nature
  - Doesn't fit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

MIDDLEWARE PROTOCOLS - 2

- **Distributed commit protocols (A2)**
  - 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

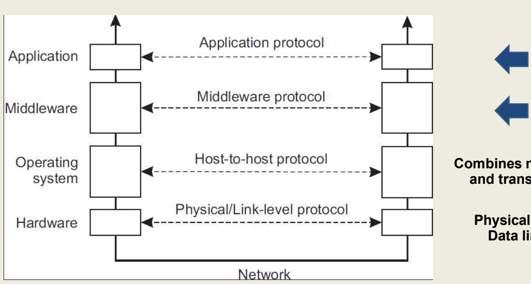
MIDDLEWARE PROTOCOLS - 3

- **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



Combines network and transport

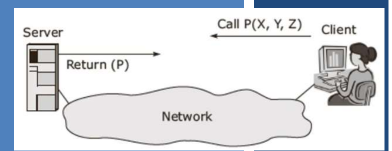
Physical and Data link

TYPES OF COMMUNICATION

- **Persistent communication**
  - Message submitted for transmission is stored by communication middleware as long as it takes to deliver it
  - Example: email system (SMTP)
  - Receiver can be offline when message sent
  - Temporal decoupling (delayed message delivery)
- **Have you ever received a back dated email?**
- **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 (**long request, large data payload**)
  3. Sender waits until request is processed and result is returned (**full**)
- **Persistence + synchronization**
  - Common scheme for message-queueing systems
- **Consider each type of blocking (1, 2, 3). Are these modes connectionless (UDP)? connection-oriented (TCP)?**



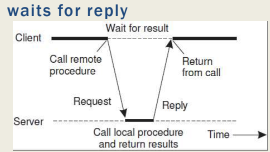
CH. 4.2: RPC

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

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*	

RPC: PASS-BY-REFERENCE

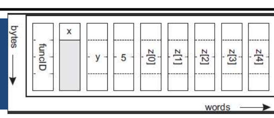
- Passing by value is straightforward
- Passing by reference is challenging
- Pointers only make sense on local machine owning the data
- Memory space of client and server are different
- Solutions to **RPC pass-by-reference**:

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

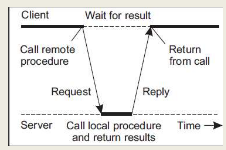
LANGUAGE BASED SUPPORT

- 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

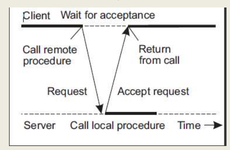
RPC VARIATIONS

- 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

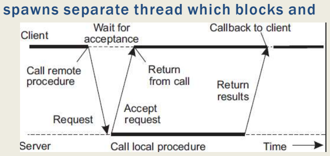


RPC VARIATIONS – 2

- 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

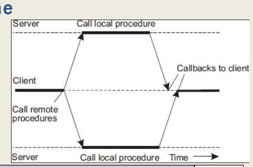
- **Deferred synchronous RPC**
  - Server performs ***CALLBACK*** to client
  - Client, upon making call, spawns separate thread which blocks and waits for call



- **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

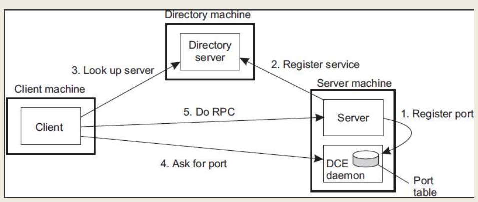
DCE – CLIENT/SERVER DEVELOPMENT

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

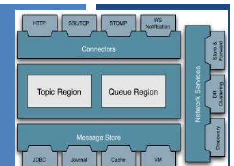
DCE – BINDING CLIENT TO SERVER

- 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



Apache ActiveMQ

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*)

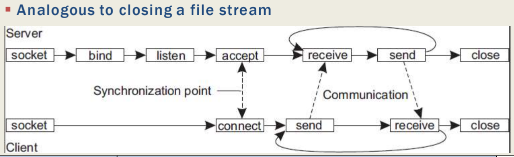
Operation	Description
socket	Create a new communication end point
bind	Attach local address to socket (IP / port)
listen	Tell OS what max # of pending connection requests should be
accept	Block caller until a connection request arrives
connect	Actively attempt to establish a connection
send	Send some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

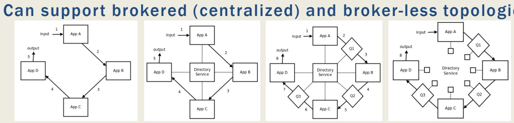


SOCKET COMMUNICATION

- 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

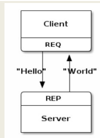


ZEROMQ – 2

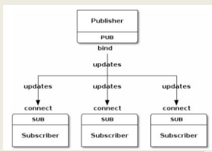
- ZeroMQ is **TCP-connection-oriented communication**
- Provides socket-like primitives with more functionality
  - Basic socket operations ***abstracted*** away
  - Supports many-to-one, one-to-one, and one-to-many connections
  - ***Multicast*** connections (one-to-many – single server socket simultaneously "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**)

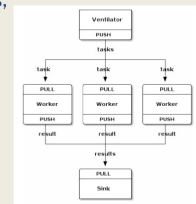


- **Publish-subscribe pattern**
  - Clients **subscribe** to messages **published** by servers
  - As in event-based coordination (Ch. 1)
  - Supports multicasting messages from server to multiple
  - Client: subscribe socket (**SUB**)
  - Server: publish socket (**PUB**)



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

QUESTIONS