CmpE 275

Section: Everything and the Kitchen Sink

Introduction to messages (v1.03)

Sequential, Spatial, and Temporal Decoupling – meh…right?

On track

- Agenda (baseline of all distributed computing)
  - Reverse Lecture 1 (RL1), Lab 2
  - On distributed processes (messaging)
    - Synchronous network systems (messaging)
    - Asynchronous systems
    - Example: Coordination through Leader Election Algos
  - Lab 1 Review
- Key points
  - Synchronous and Async
  - Message queue
  - Sequential / Spatial / Temporal decoupling

Reverse Lecture 1 Topic

RL1 is more guided for initial experience. Whereas, RL2/3 are unstructured

Multi-process (3) message passing to maximize voracity, volume, velocity while minimizing latency.

Given:

- 3 Processes (A, B, C)
- Point-to-point (A – B – C)
- Measure/Validate to defend your design
- Language: Java and/or Python
- Run through shell scripts
- Low-level (basic) third-party libraries only (no Kafka, Spring, or other similar toolkits)
- Bonus score for the fewest third-party libraries used

For the next lab (2): Beyond metal
(Data Representation)

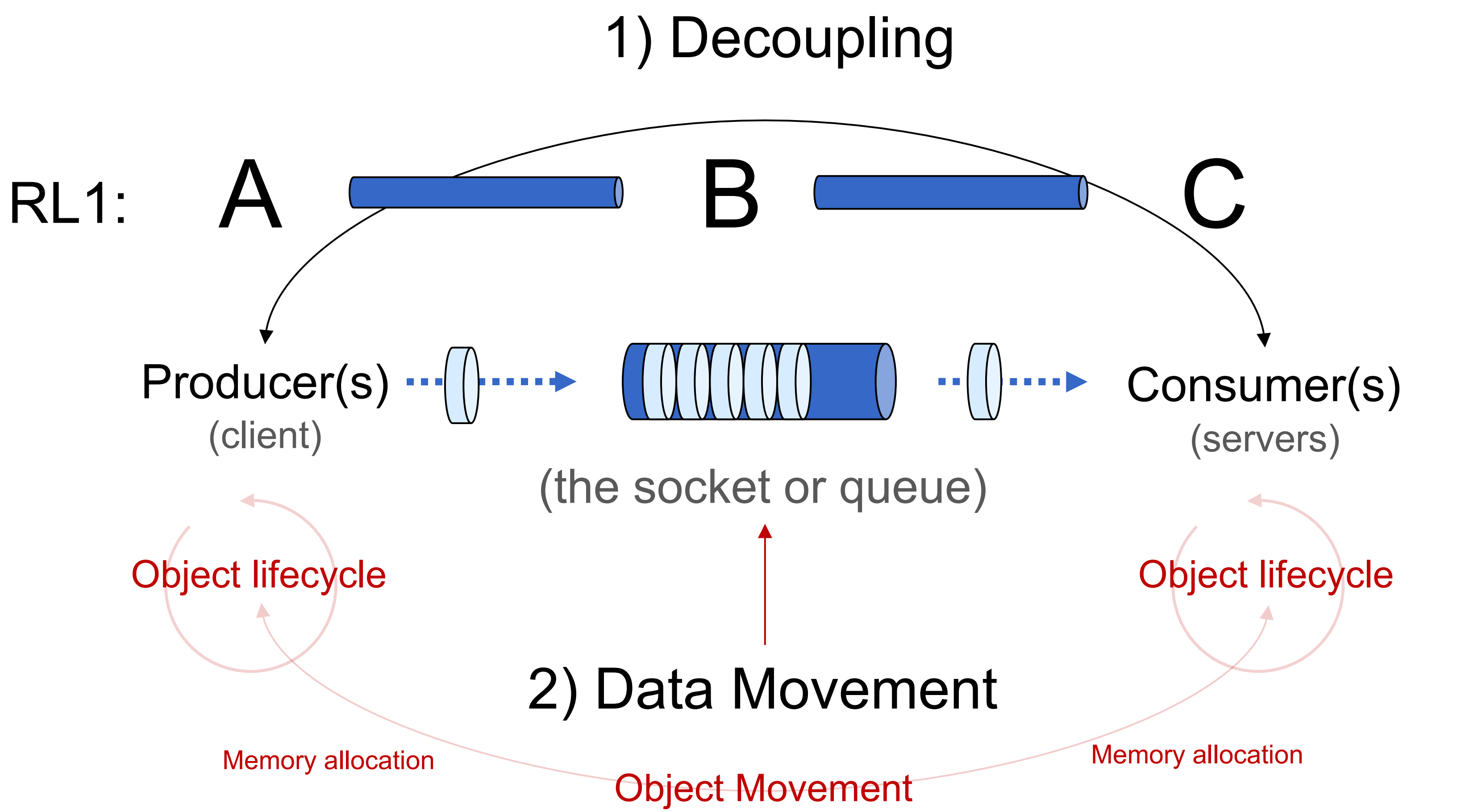
- The basic mechanics of the socket lab introduces challenges that we will explore in depth. In the bare-metal we constructed a single client-server (two-tier).

- Lab 2 focuses on movement and memory
  - Data representation
  - Long chains, cycles, and congestion
  - Software – Abstraction vs. complexity
    - more features, more implementation hiding
  - Failure and recovery – lost connections, data
  - Managing: Velocity, Voracity, Volume – the three-Vs

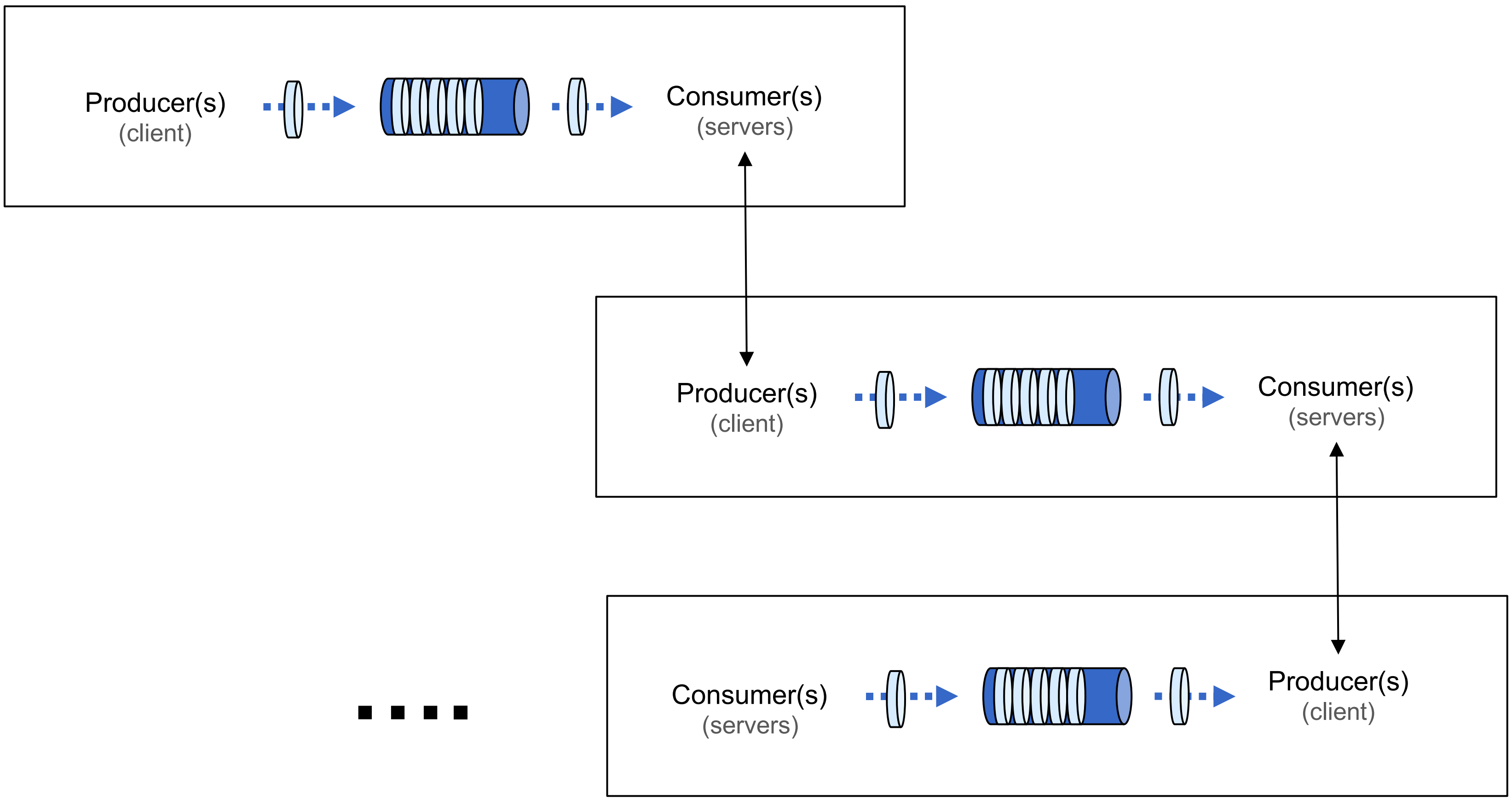
What use cases are there for distributed computing (**Messaging** and **Queuing**)?

- Bridging Between Applications
  - Async/Sync system integration (Proxy/Façade)
    - E.g., An online business to a payment gateway
    - E.g., Legacy system integration
- Intermittent Communication
  - Delivery of content on faulty networks
  - E.g., Faulty networks
- High Volume, High Speed
  - Bandwidth and memory constraints
  - E.g., Streaming content & data movement

Bare Metal lab focuses on the 2 fundamentals of communication



Pairwise (P–C) components are connected to create complex (n-tier) relationships



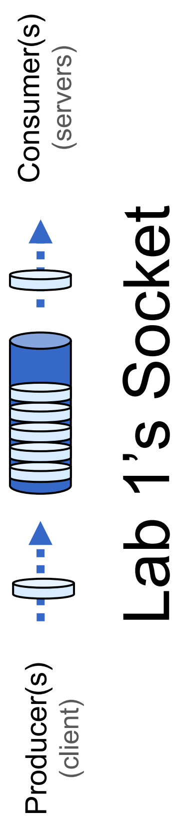
Challenges (limited by)

- Our goal
  - Manage time
  - Manage memory
  - Manage CPU cycles
  - Manage bandwidth

Pressure or limitations of the code is where we find challenges of scale and performance in our design

- So how do we do this?
  - FASTER computers?
  - Better networks?
  - Great software?

Message-based communication



The Message Queue (MQ) concept

- Forms: Sockets, web services, JMS, reading/writing files, async HTML, …

So, what is MQ?

- What are the concepts?
- What functionality does it [MQ] provide?
- How do I use these concepts?

Recall

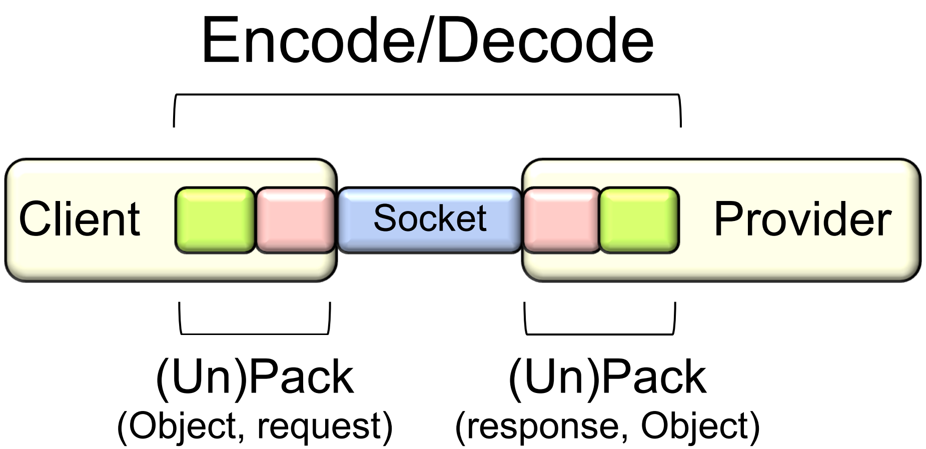
1. Decoupling
2. Movement

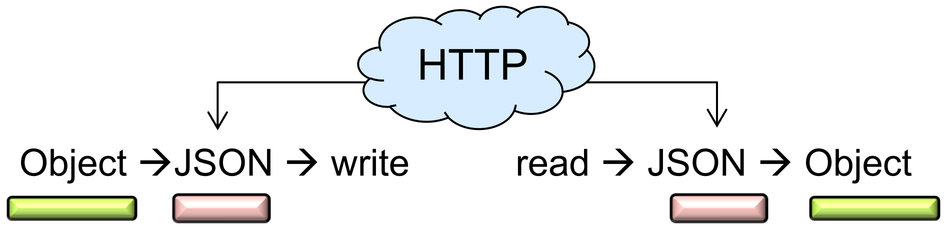
Complex data: Overloading

- How to represent data (encode)?
  - Attachments (e.g., Multi-part)
    - Data is stored as a separate document attached to the request (upload, download a file) or referenced URL (common storage)
  - Name-value pairs (attributes, parameters), tuples
    - Form-like (e.g, firstname-value, lastname-value)
    - Converting to name-value pairs is difficult if the data is complex and/or deeply nested – representing graphs of data (hierarchical), the data is flattened (row-column)
    - How to support inheritance? Data changes (releases)?

Data (Payload) Representations
(E.g., Binary, JSON, XML, KVP, Text, HTML)

- Delivering complex data structures using <u>overloading</u>
  - Type overloading
    - application/json, application/xml
    - Encoding data for the client and/or server to process
  - Value overloading
    - Overloading POST or PUT





Increases complexity because the data is tunneled to the server through the form parameters.

Data Movement

- Blocking
  - Caller waits for the response
  - E.g., Calling a method/procedure

```
var data = myStorage.loadData();
var text = data.toString();
```

- Non-blocking
  - Caller does not wait
  - Sequence, Spatial, and Temporal

Sequence decoupling

- Parallel (Sequence decoupling)
  - Parallelization of a request into smaller components that can be processed concurrently.
  - Behavior is both synchronous and asynchronous
  - Example: Data decomposition
  - Advantages
    - Processing large amount of information that otherwise would be difficult or inefficient in a serial algorithm

Spatial decoupling

- Location/Space (Spatial decoupling)
  - Interactions are not limited to the current process space, computer, system
  - The client and server are not required to co-exist on the same server, OS, and language
  - Advantages
    - Server-side scaling – architecture can change without affecting the producer

Temporal decoupling

- Time (Temporal decoupling) – asynchronous behavior also frees processes to act
  - No timing dependencies between producer (client) and consumer (server). The consumer is not required to act immediately when a message is produced
  - Advantages
    - Consumer can defer processing a message. This allows the consumer to apply QoS and fair scheduling practices
    - Partially supports a partitioned network (why only partially?)

Types of distributed computing: parallel and concurrent?

- Parallel
  - simultaneous, independent execution of tasks
- Concurrent
  - scheduled cooperative (interleaved) execution where (typically) only one thread is active.
- Distributed
  - Parallelization across processes in asynchronous designs

What are the incentives to use distributed architectures?

1. Scaling
   a. CPU
   b. Memory
   c. I/O

2. Failure-Recovery

3. Performance

4. Isolation

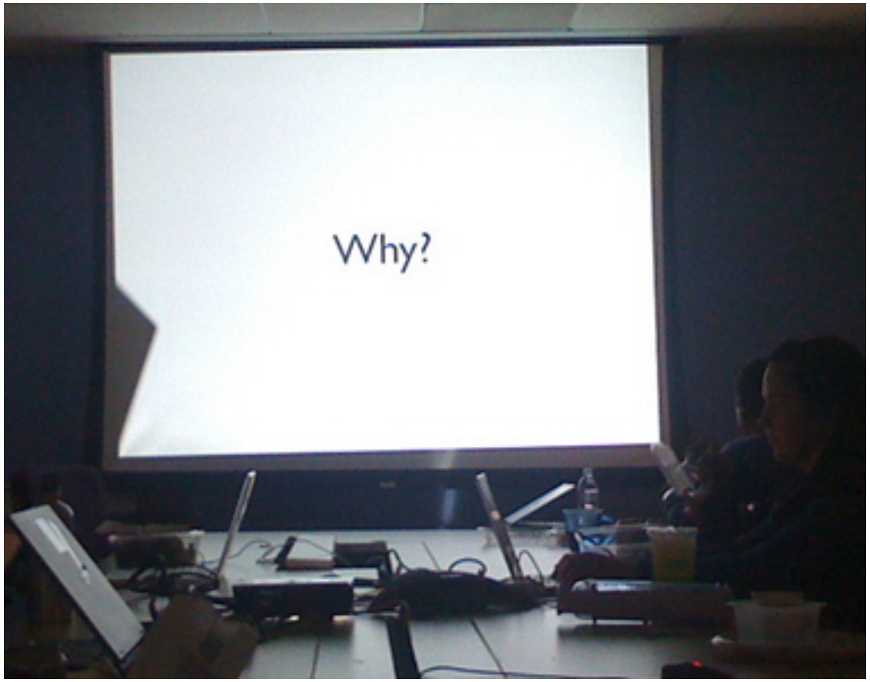


Image: Why? by Myles! on Flickr

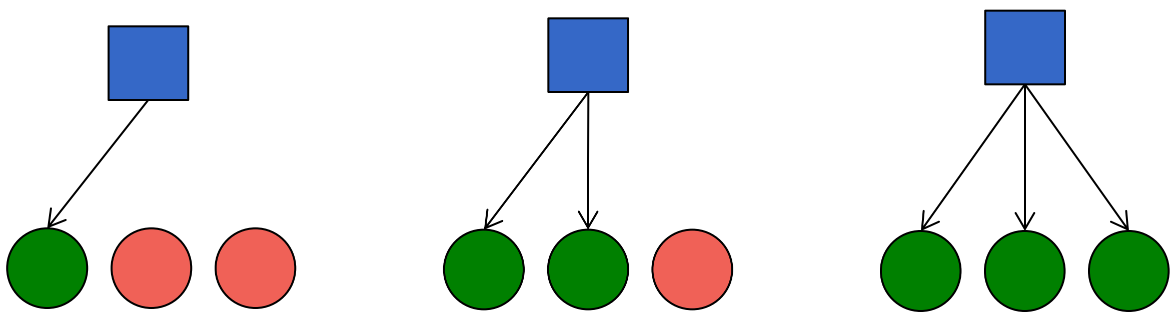
Recall complex applications are built with pairs

- The connection (Edge) between two processes (Nodes A,B → AB) are the core structures in which threads and processes scale to complex systems.

- Core patterns are built upon
  - Message passing patterns
  - Queuing behavior
  - Overlay network design/construction

Review: Types of message passing

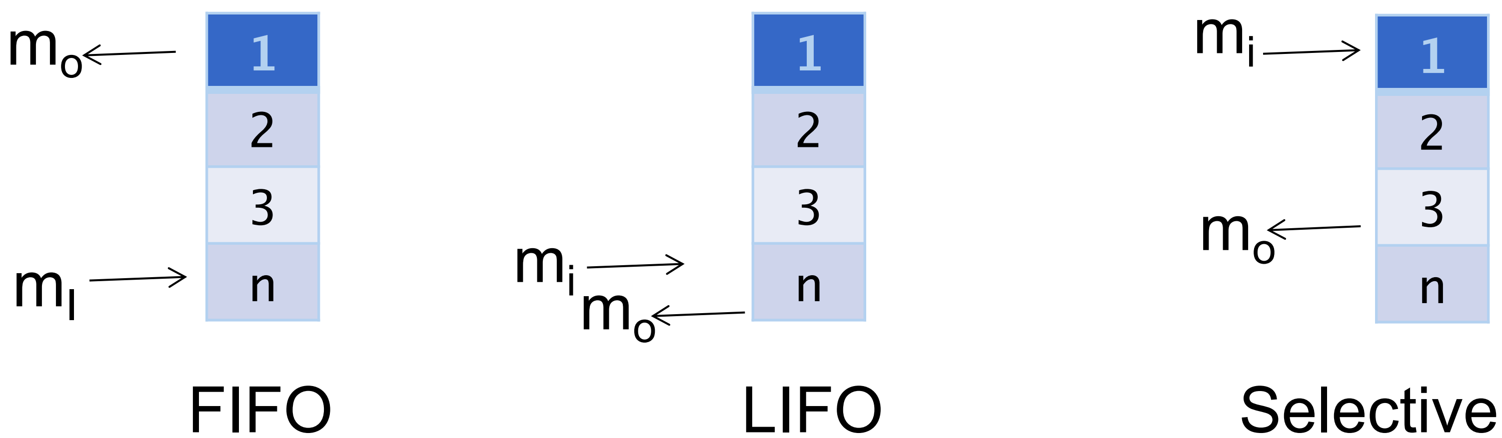
- Message passing generally falls to the following patterns



pt-to-pt Multicast Broadcast

Review: Queue behavior: FIFO, LIFO, Selective

- Enqueue and dequeue behavior of stacks or queues



Question: How would you implement a multi-process design for each?

Review: Overlay networks

- Overlay networks are logical constructions of physical assets (computers, processes) that represent the architectural features like
  - Data or temporal space (e.g., sharding and archives)
  - Functional decomposition (e.g., workflow)
  - Logical decomposition (e.g., MVC)

Examples of overlay network organizations

- Rings
- Hubs
- Bus
- General Graphs
- Proxies
- . . .

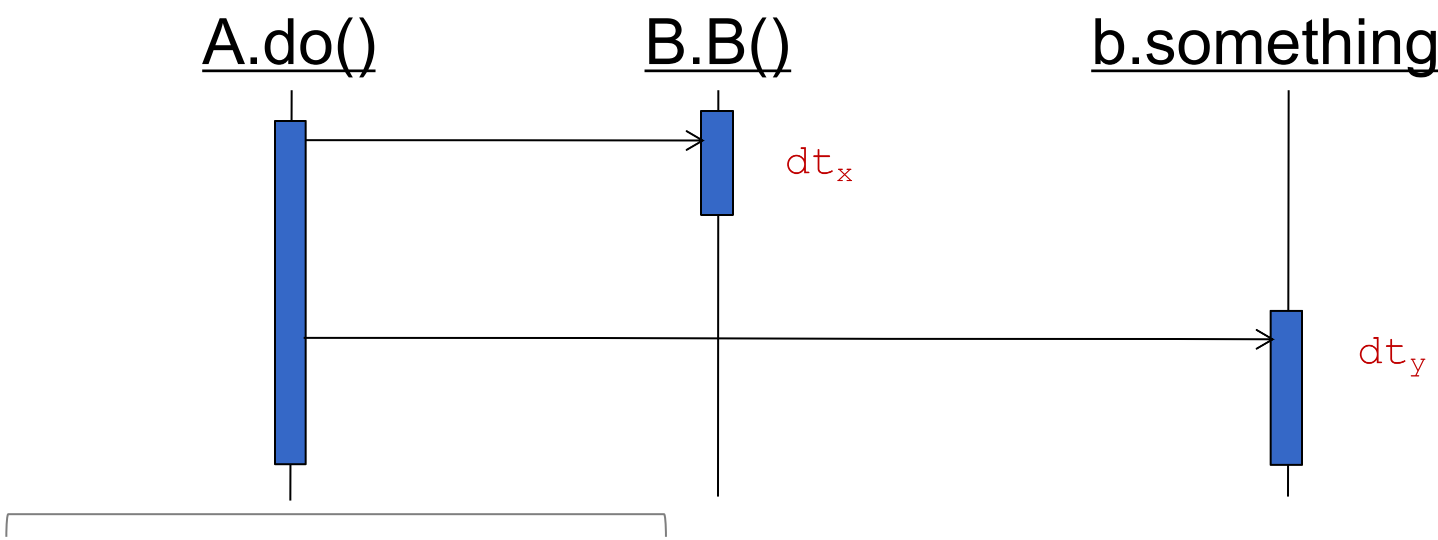
How do these designs affect communication?

Asynchronous Messaging

Thinking beyond the simplistic 2-Tier (Client-Server) models

Synchronous message processing is similar in behavior between two classes

- From within a method (same process)
  - E.g., `A.do() { b = B.new(); b.something(); }`

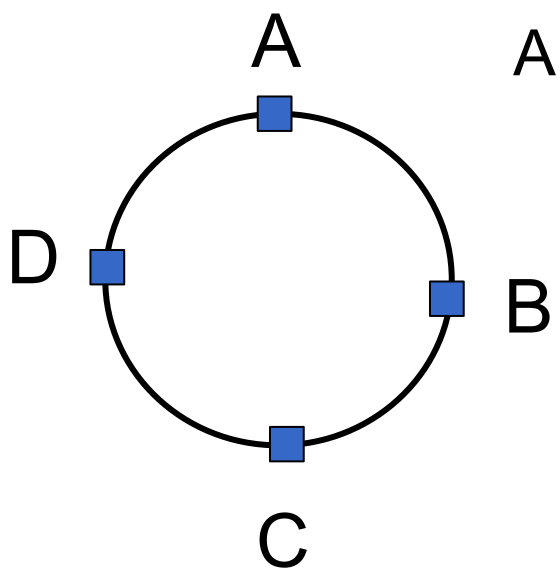


$$dt = dt_x + dt_y + C_{Latency}$$

if dt << acceptable liveliness (AL) then the solution is good. **What if dt >> AL?**

Consider a ring network in synchronous (blocking) communication

- Given 4 Nodes (A, B, C, D)
- How does A send a message to D?

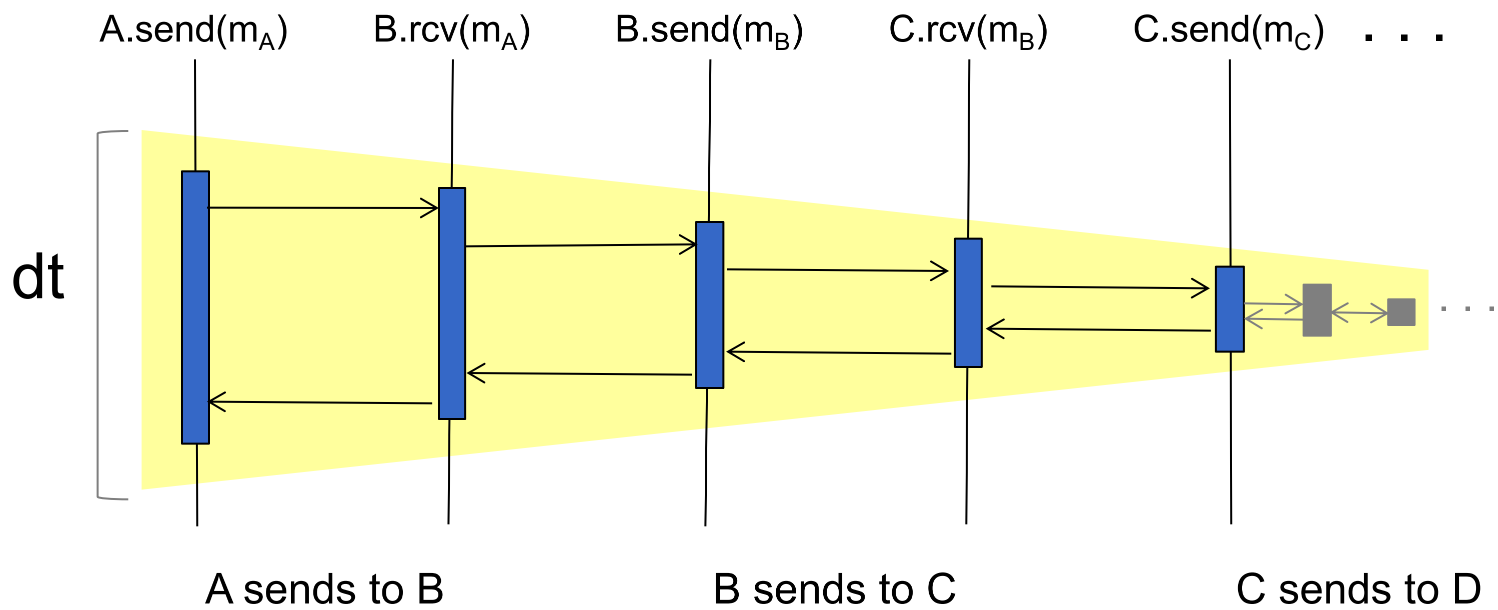


A.send(B,m)

B.getMsg(m) { B.send(C,m); }

. . .

Synchronous chained messaging does not have a constant dt ($dt = \Sigma\ (dt_i + Latency_i)$)



$dt_{Liveliness} = dt_A + dt_B + dt_C + dt_d + (6 * Latency)$

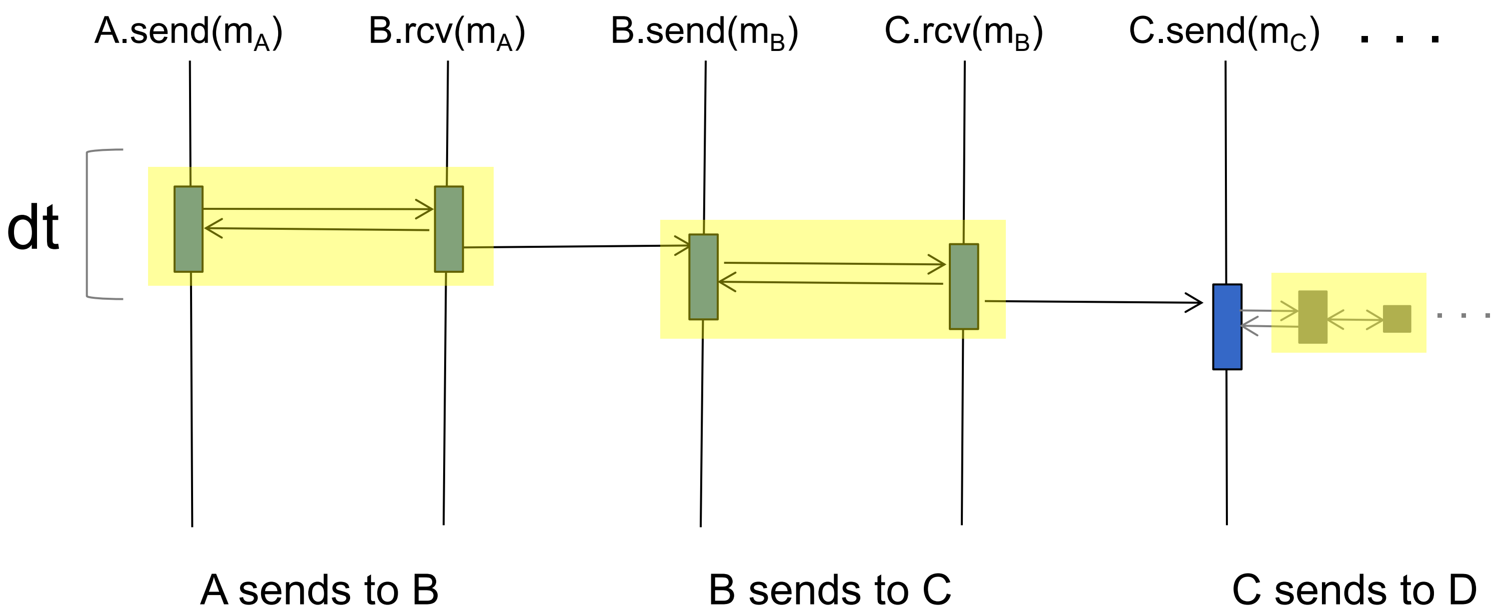
Technologies utilizing synchronous messaging include HTTP, Java RMI, CORBA, gRPC, CUDA, OpenMPI, Java EE (EJB), Web Services, and DBMS

Asynchronous (Non-blocking) inter-process communication provides approx. constant dt

$(dt = dt_i + Latency_i)$

- Liveliness from Node A's perspective

$$dt_{Liveliness} = dt_A + (1 * Latency)$$



For Consideration: Synchronous vs. Asynchronous problem solving

- Given
  - 5 nodes (processes) arranged in a ring
  - Each node is equidistant (1 m) from each other
  - A complete request uses all nodes (A-B-C-D)
  - A node can only process one request at a time
  - The latency between nodes is 250 msec
  - Nodes A and C are Intel XEON (4 cpu 8 core servers)
  - Nodes B and D are AMD (2 cpu 12 core servers)
  - Each node requires 1 min 30 sec to process a request

- For asynchronous and synchronous calculate

  1. What is dt for node A?

  1. How long does it take the network to process a request?

Approaches associated with asynchronous messaging

- In the previous problem, a process could only operate on a single message at a time.
  - In a synchronous configuration, this was okay as blocking 'fit within' the design

- For asynchronous messaging (non-blocking requests), how do we provide single message processing while accepting multiple requests?

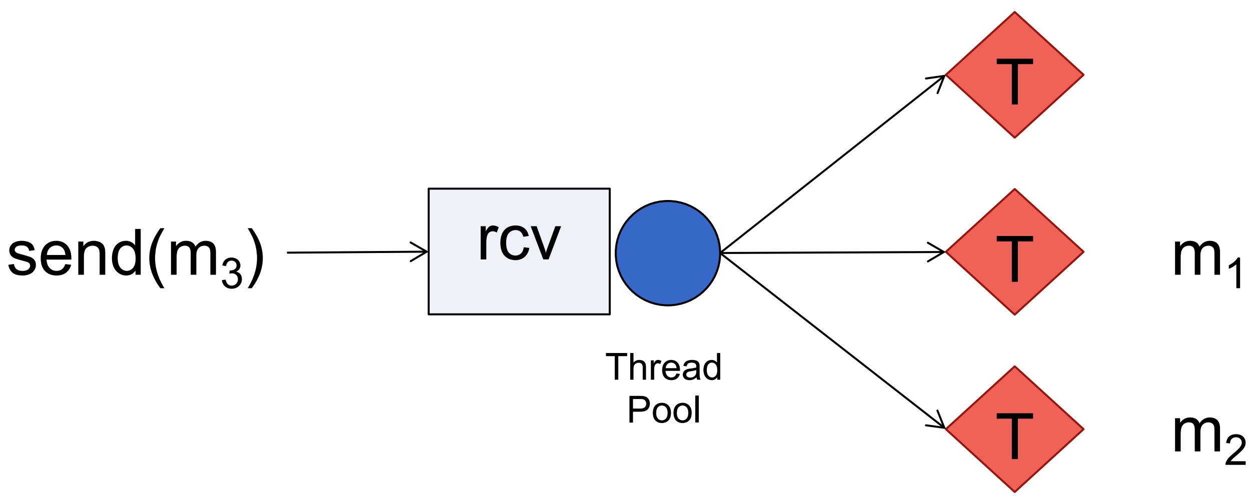
Socket-per-thread

- If our server receives messages and for each message creates a thread
  - Like our socket-toolkit – right?

- Okay, so what is the concern with a thread per request?

Socket to thread pool solution

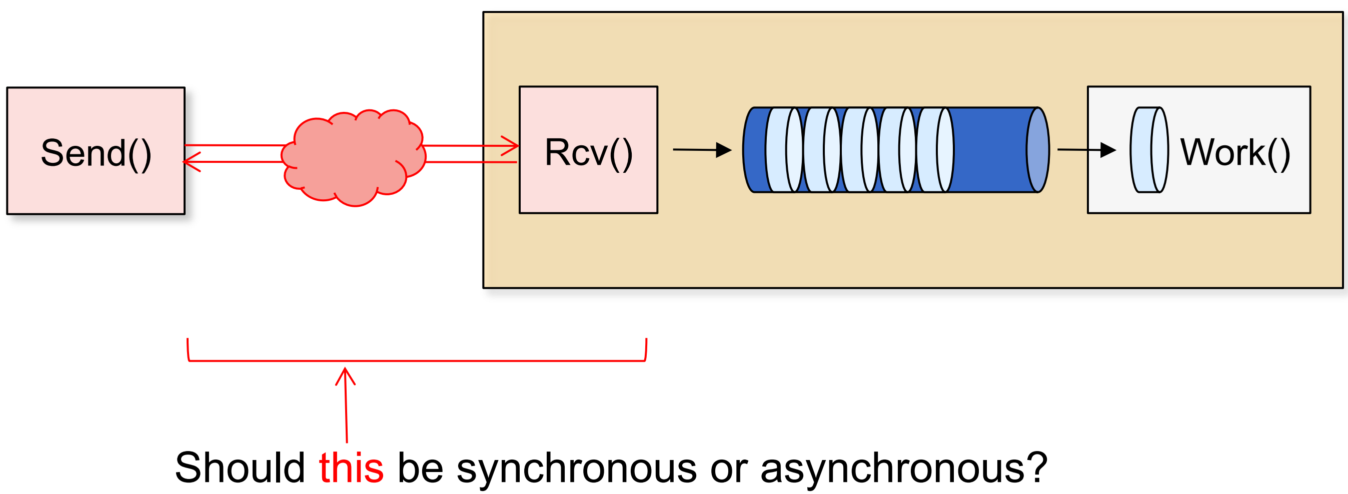
- Limit the number of threads to minimize thread creation/execution



What about m_4 , m_5 , m_j ?

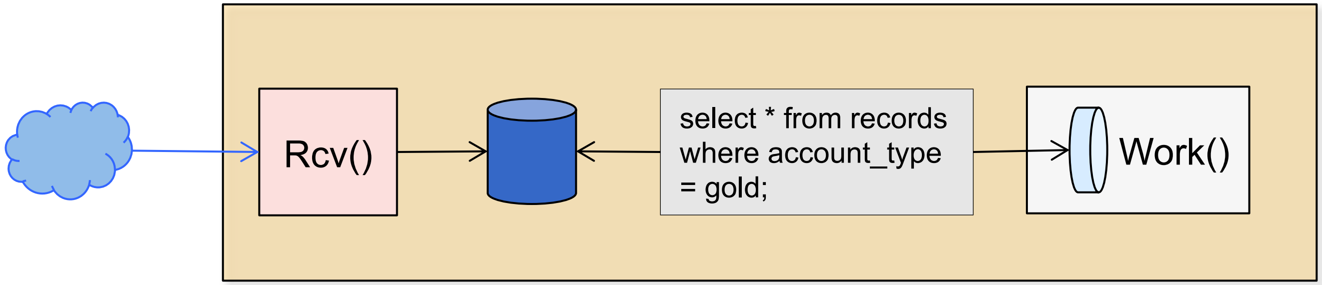
Queuing of inbound messaging provides a buffer between the receiving process (doing the work) and the sender (client requesting the work)

- Queuing of requests provides
  - Preserves client's asynchronous ability without loosing requests
  - Receiving (Server) can process messages to its ability; rate (i.e. messages/sec)



Enhancement 1: dequeue query to provide selective processing

- enqueued requests in a basic queue acts as a FIFO
  - PRO: request processed as received, this will support passive sequencing (no special processing for sequentially dependent data
  - CON: No (built-in) selective processing
    - E.g., prioritization of messages
- Query queue for pre-processing prioritization
  - Implementations can range from volatile memory to persistent message storage (use of SQL)

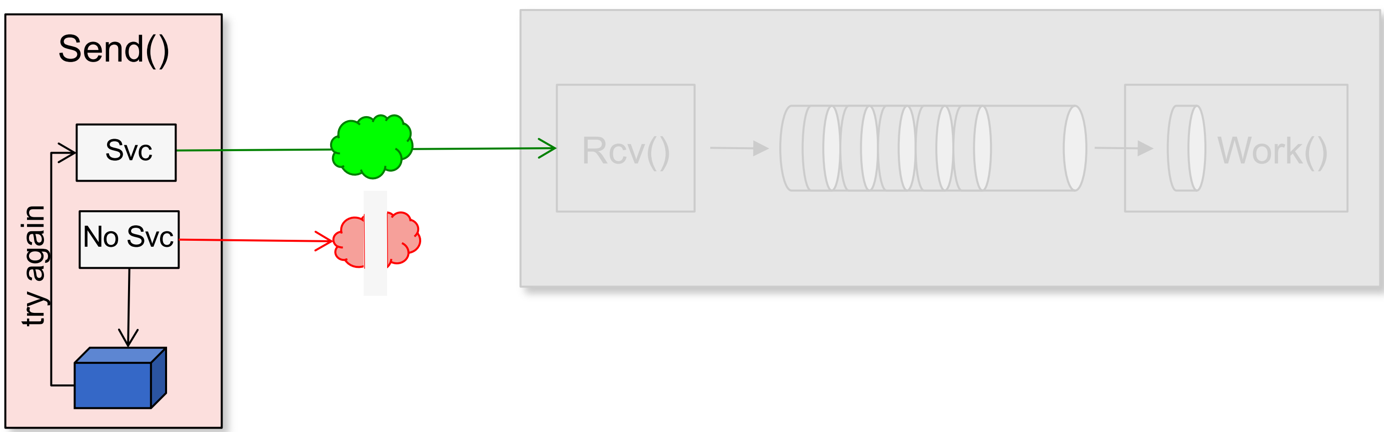


Incidentally, how do you imagine JMS works?

Enhancement 2: client-side or middleware queuing (store-and-forward)

- Store-and-forward is the ability for the client-side (sender) to store messages if the receiver (server) is not available
  - From the client's perspective the message was sent (no action required under the assumption of asynchronous behavior)
  - Questions:
    - How would a synchronous communication fair?
    - What are the risks of this approach

Upon re-establishing a connection the client stored messages are sent to the receiver/server



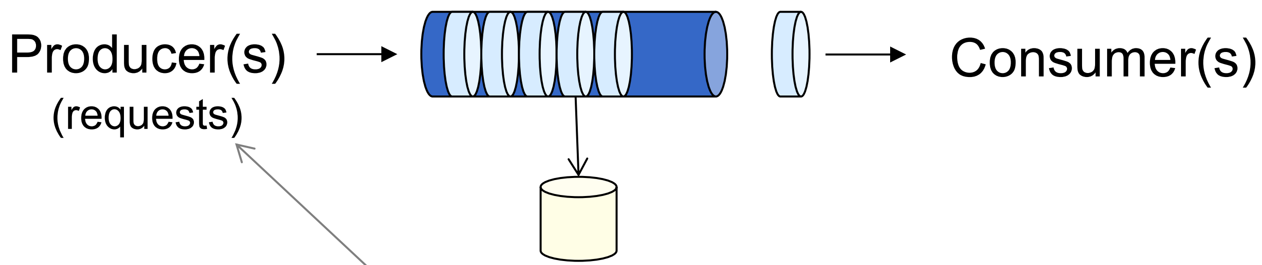
Local message persistence

Enhancement 3: Quality of Service (QoS)

- What do we mean by QoS?

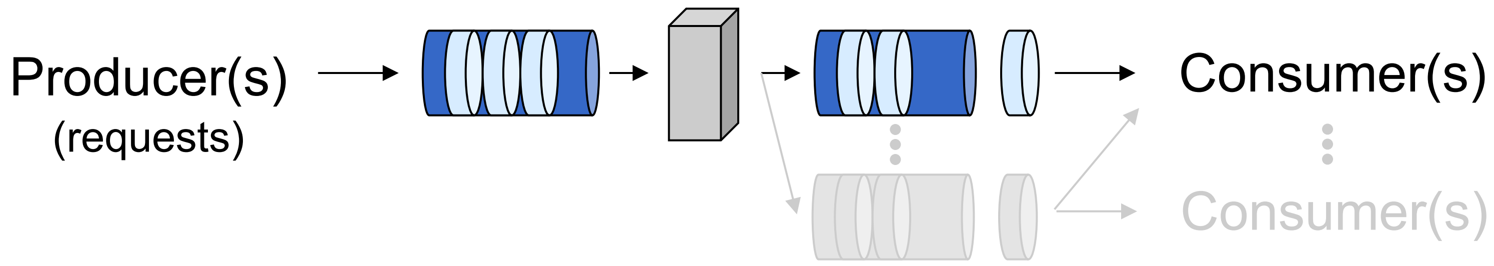
- What is our QoS design?

QoS: Simple queuing



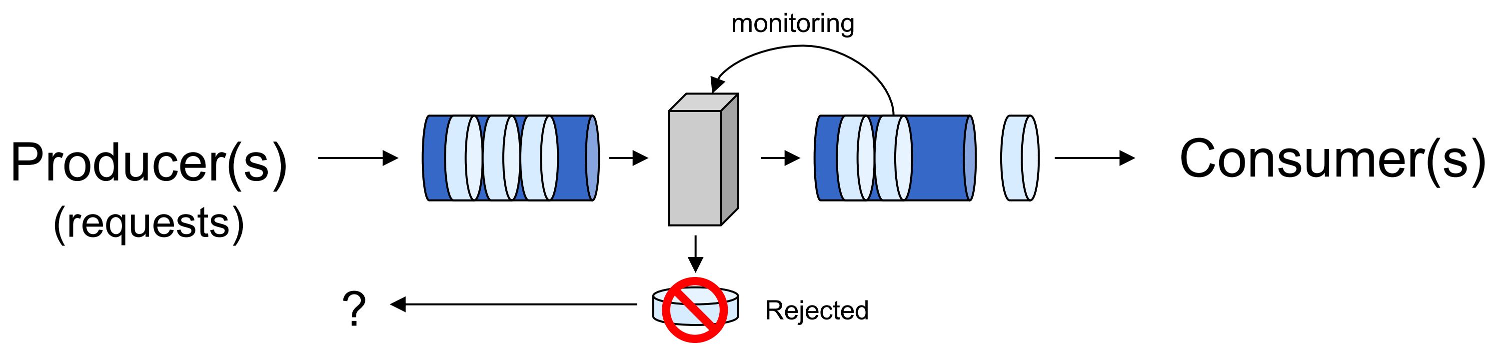
- Simple queue
  - "Best effort" or "Store-and-Forward"
  - Provides buffering between producer and consumer (overflow)
  - Simple to step up, use, and support under the right conditions
  - FIFO behavior
  - Limited to no control of resources – reactive or push design
    - lacks prioritization of requests
    - Prolific producers can starve other producers (clients)
  - Variants
    - Durable – storage provides buffer overflow and persisting for failure/recovery
    - Consumer pool – multiple consumers (I.e., MDB)

QoS: Fair Queuing



- Fair Queuing (FQ) and variants
  - Another “Best effort” QoS
  - Use of multiple queues to bin requests
  - Simple spraying or overflow binning across multiple queues
- Variants
  - Employs simple algorithm(s) to determine which queue (bucket) to place a message
  - Weighted Fair Queuing (WFQ)
  - Hierarchical Weighted Fair Queuing (HWFQ)
  - Class-based WFQ (CBWFQ)

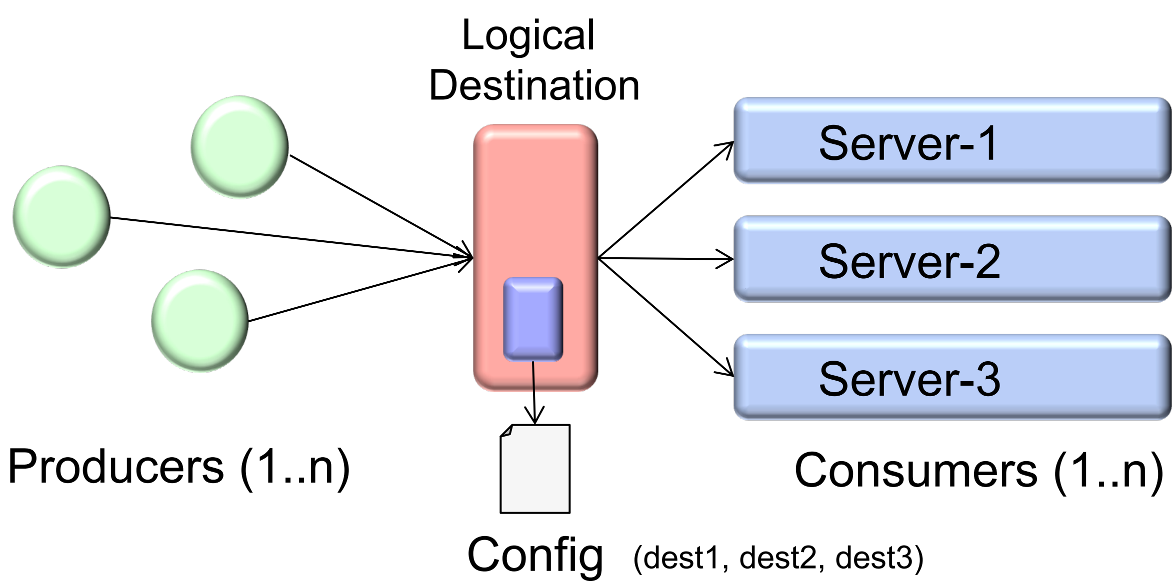
QoS: Early detection (rejection)



- Random Early Detection (RED)
- Variants
  - Weighted RED (WRED)
    - Moves buffering and rescheduling onto the producer (client)
    - Drop messages when unable to process
    - Weighted – apply criteria (producer, priority, etc) to drop
    - Generally, not a friendly approach
  - Class-based WFQ (CBWFQ)
  - RSVP or Guaranteed service (hard QoS)
    - Reservation of queue

Enhancement 4: Failover and scaling

- What is happening here?
- PROs and CONs?



Other enhancements

- Concurrent/Parallel message processing
- Quality Control (pre-filtering)
- Dynamic payload sizes (optimizing throughput)
- Sequencing (Ordering) of messages
- Authorization and bandwidth management
  - QoS application
- Auditing
- Encryption and compression

Many F/OSS and COTS packages

- Implementations
  - OpenJMS (http://openjms.sourceforge.net/) – low activity since 2007
  - NATS cloud mq – https://nats.io
  - SwiftMQ (http://www.swiftmq.com)
  - RabbitMQ, an AMPQ implementation
  - ZeroMQ (http://zeromq.org)
  - Apache
    - Qpid – AMPQ implementation
    - ActiveMQ (http://activemq.apache.org/)
  - Tibco broker (http://www.tibco.com)

Reading and references

- Papers and discussions
  - http://bravenewgeek.com/dissecting-message-queues/
  - http://www.dynamicobjects.com/papers/w4spot.pdf
  - https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/50b7ac8d-0aed-2a10-d290-b64f44c4c1a9
  - http://www.precisejava.com/javaperf/j2ee/JMS.htm

- Software used in to support the messaging lab (distributed off-line)
  - RabbitMQ – http://www.rabbitmq.com
  - Erlang – http://www.erlang.org

Reading and references

- JMS
  - http://72.5.124.55/j2ee/1.4/docs/tutorial-update6/doc/JMS3.html
  - JMS Specification, version 1.1 available from
    http://java.sun.com/products/jms/docs.html

- Consensus
  - Distributed Algorithms, Nancy Lynch, 1996
  - Using Paxos to Build a Scalable, Consistent, and Highly Available Datastore, Rao, Shekita, Tata, 2011
  - A Survey of Consensus Problems in Multi-agent Coordination, Ren, Beard, Atkins, 2005
  - The Byzantine Generals Problem, Lamport, Shostak, Pease, 1982
  - Paxos Made Simple, Lamport, 2001
  - Paxos Made Live – An Engineering Perspective, Chandra, Griesemer, Redstone, 2007

Appendix: JMS Summary (Historical)

Concepts and Features

Java (Jakarta) Messaging Service (JMS)

- Jakarta (Java) Messaging Service (JMS)
- JMS version 2.0 (2013) – JSR 914 (v2.1, 2003)
  - Originally a stand-alone package, integrated into J2EE. Transitioned to Java Community Process
  - Implementations: ActiveMQ, Amazon SQS, Apache Qpid, RabbitMQ
  - Significant restructuring to simply API
  - Domain unification
    - Simplified API for general, reusable code
    - Common interface allows the same class to send and receive messages
    - Unification allows a session within a single TX to send and receive
    - Session can manage queues and topics

Asynchronous-ish messaging

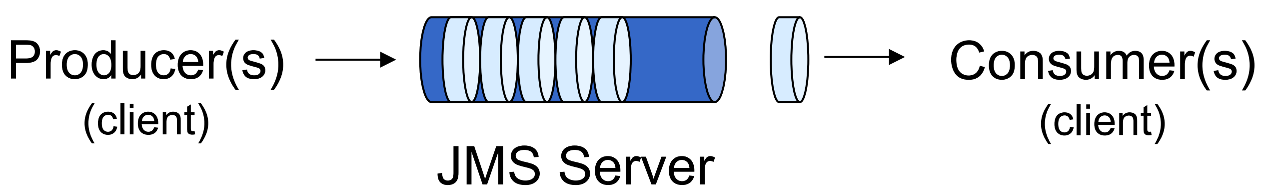
- JMS defines asynchronous messaging as independent request and response between the producer and consumer. The consumer can choose between blocking and event processing.

- Producer (sender)
  - Non-blocking – `MessageProducer.send()`
- Consumer (receiver)
  - Blocking
    - `MessageConsumer.receive()`
    - Consumer blocks on the receive() waiting for messages or time out
  - Non-blocking
    - `MessageConsumer.setMessageListener(MessageListener l)`
    - Implemenation of javax.jms.MessageListener interface is invoked upon receipt of a javax.jms.Message
    - This is how MDBs are defined in Java EE

```
public class MyListener implements MessageListener {
    public void onMessage(Message message) {…}
}
```

JMS Queues

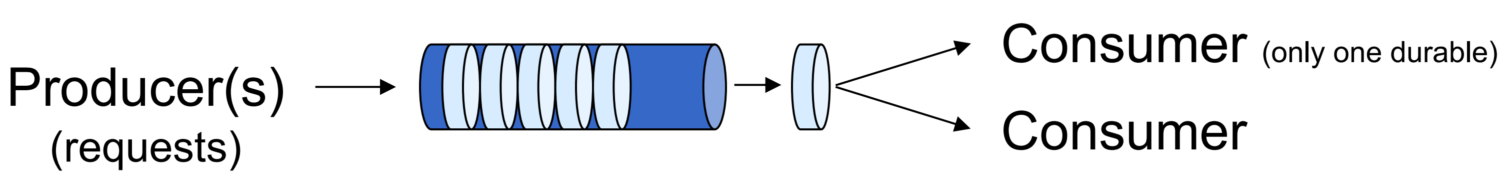
- Point-to-Point (PTP)
  - Each message is delivered to one consumer
  - If no consumers are registered the message is held unless an expiration was set – `MessageProducer.setTimeToLive(msec)`
- `javax.jmx.QueueBrowser`
  - Look at messages without removing them
- Sequence
  - Producer connects to a message queue
  - Producer sends message (blocking)
  - Returned successful – message delivered to JMS server
  - If no consumers are listening, the message is held until a consumer connects
  - If more than one consumer is registered, the message is only delivered to one consumer



JMS Topics

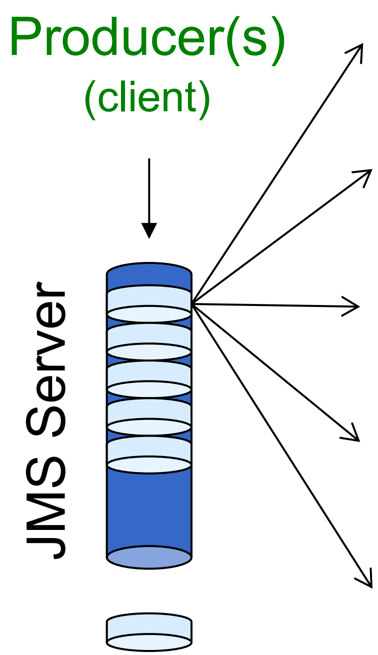
- Publish/Subscribe messaging (Fanout or Broadcast)
  - Each message can have multiple consumers (same message will not be delivered twice*)
  - Messages are guaranteed to be delivered to any consumers if a consumer's subscription is durable. Otherwise, they are discarded.

```
Session.createDurableSubscriber()
```



**Unless transactional control is implemented and a rollback is performed*

JMS supports the transport of many payload representations



- TextMessage
  - Text (a.k.a String)
- ObjectMessage
  - Serializable object
- StreamMessage
  - Sequence (FIFO) of java primatives
- BytesMessage
  - Stream of uninterpreted bytes
- MapMessage
  - Send name-value pairs
  - Names are String objects and values are primitives

Example: Queue producer

```
// OpenJMS
Context context = null;
Connection connection = null;
try {
     context = new InitialContext();
     ConnectionFactory factory =
         (ConnectionFactory) context.lookup(factoryName);
     Destination dest = (Destination) context.lookup(destName);
     connection = factory.createConnection();

     Session session =
          connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

     MessageProducer sender = session.createProducer(dest);

     connection.start();

     TextMessage message = session.createTextMessage();
     message.setText("Hello");
     sender.send(message);

} catch (Exception exception) {
     . . .
} finally {
     context.close();
     connection.close();
}
```

Example: Queue consumer

```
Context context = null;
Connection connection = null;
try {
     context = new InitialContext();

     ConnectionFactory factory = (ConnectionFactory)
     context.lookup(factoryName);
     Destination dest = (Destination) context.lookup(destination);
     connection = factory.createConnection();

     Session session = connection.createSession(false,
     Session.AUTO_ACKNOWLEDGE);

     MessageConsumer receiver = session.createConsumer(dest);

     connection.start();

     Message message = receiver.receive();
     if (message instanceof TextMessage)
          TextMessage text = (TextMessage) message;

} catch (Exception exception) {
     exception.printStackTrace();
} finally {
     context.close();
     connection.close();
}
```

Should be contained in a loop: while (true) {…}

A consumer that is event driven

```
// create the connection
connection = factory.createConnection();
session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);

MessageConsumer receiver = session.createConsumer(dest);

// register a listener
receiver.setMessageListener(listener);
```

```
public class MyListener implements MessageListener {
   public void onMessage(Message message) {
      if (message instanceof TextMessage) {…}
   }
}
```

Message delivery is not guaranteed (perspectives)

- Message delivery mode
  - JMS supports two delivery modes
    - **NON_PERSISTENT** – Does not requre the JMS server to ensure message delivery
      - Delivery is **at-most-once** (which includes zero)
    - **PERSISTENT** – JMS server (provider) is instructed to ensure message is not lost between producer and consumer
      - Delivery is guaranteed **once-and-only-once** (successful/acknowledged)
      - However, the JMS server may “experience resource limitations” that could result in lost messages
- Message’s Time-To-Live
  - A producer can specify how long a message should be retained before discarding
  - A discarded message is not delivered (persistent or non-persistent)

Acknowledgement modes when sending messages

- The three acknowledgement modes are
  - AUTO_ACKNOWLEDGE
    - Automatically handled by the Session when
      - send() or receive() returns
      - Call to the `MessageListener.onMessage()` is called
  - DUPS_OK_ACKNOWLEDGE
    - Lazy session acknowledgement
    - Duplicate acknowledgements may be sent
  - CLIENT_ACKNOWLEDGE
    - Explicit control of acknowledgement, consumer controlled – allows for batch processing and rollback

The JMS Message class

- javax.jms.Message
  - Composed of
    - Header – methods beginning with setJMS and getJMS
    - Properties – get/setXXXProperty()
    - Body – implementation specific (e.g., setText(), getText())

Sound familiar?

Transacted sessions

- Transacted sessions are used with CLIENT_ACKNOWLEDGEMENT sessions to confirm or reject message handling
  - `Session.commit()`
  - `Session.rollback()`

- How it works
  - For a producer, messages are not sent to a consumer until commit() is called
  - For a consumer messages are not confirmed received until commit() is called

Advantages and disadvantages in simple queuing

- Pros

- Cons