ASYNCHRONOUS MESSAGING

In distributed computing, one of the harder problems is scalability. For instance, a web site may have a page where users enter an order to purchase an item. The code behind the web page needs to validate and then deliver the order to a back-end system for further processing.

The following code illustrates a naïve, synchronous implementation of the code the web page may use to send this order to the back-end system. The following code uses RMI (Remote Method Invocation) to communicate with the OrderServer.

```
Registry registry = LocateRegistry.getRegistry(orderRegistry);

OrderService orderService = (OrderService)registry.lookup(orderRegistryName);

SubmitOrderResult result = orderService.submitOrder(order);
```

This code, although calling to another party in a distributed environment, executes sequentially. It is easy to write. One of the main problems is that while waiting for a response, the thread of execution is suspended; as the number of orders increases, the number of threads that wait for a response from the OrderServer increases as well. Therefore, it would be more advantageous to send the order to the order service without needing to wait. At most, a confirmation that the order has been stored for processing may be required.

Asynchronous messaging technologies are usually applied to this problem. With Java, a common API used for this is JMS (Java Message Service). JMS has several models, including a queue-like API similar to Amazon's SQS (Simple Queue Service). The following code uses JMS to communicate with the OrderServer.

```
QueueSender queueSender = queueSession.createSender(orderServiceQueue);

queueSender.send(orderMessage);
```

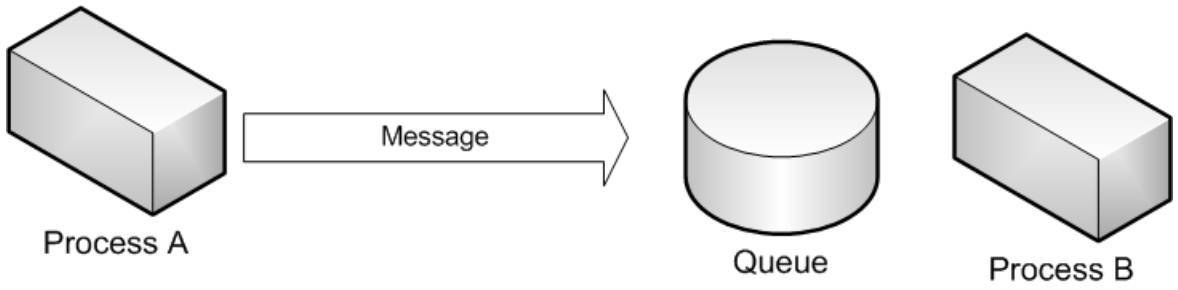
Sending to the queue with this API allows for greater responsiveness at the sacrifice of immediately obtaining order processing results. However, the underlying JMS technology, usually a transactional database, guarantees the order is not lost. A new user interface (such as a new page or AJAX-enabled portion of a page) can handle end-user requests to check on the status of the order by querying a database. Alternatively, when the order service has finished processing the order, it may send an order processing results message of its own.

```
QueueSender queueSender = queueSession.createSender(orderResultsQueue);

queueSender.send(orderResultsMessage);
```

The original sender can have code that processes messages in this result queue. This is similar to the code the order service would use.

```
QueueReceiver queueReceiver = queueSession.createReceiver(orderResultsQueue);
Message message = queueReceiver.receive();
```

Inside the message object is the order results. Figure 1 illustrates the topology of this design.



In the order service example outlined above, there are two actors: a client sending orders and the order service that processed them. Also of interest is another asynchronous messaging scenario called publish-subscribe (or pub-sub, or eventing.)

With pub-sub, an actor sends a message to any number of potential subscribers asynchronously. In this example, the subscribers have subscribed to a topic the message is a part of. In some scenarios, the first subscriber, who has resources (such as processing time), will then process the message. In other scenarios, each subscriber will process the message entirely and separately (usually each subscriber has a different task to perform in such cases).

Going back to the order processing example, it is easy to imagine that a logging service will want to log each order before any kind of processing is done to it. This is an orthogonal operation to the actual order processing and can be performed in parallel. So, we can modify the messaging pattern used to pub-sub.

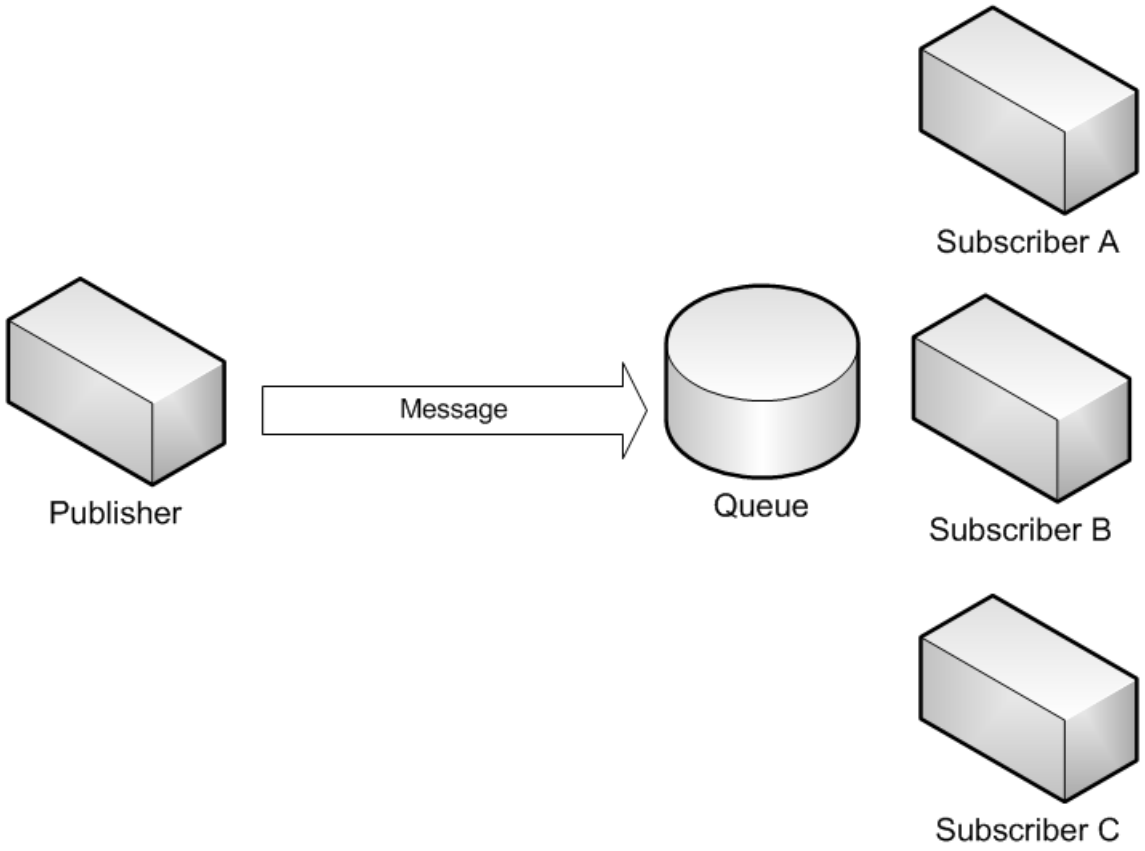
The client sends a message to a topic:

```
TopicPublisher topicPublisher = topicSession.createPublisher(ordersTopic);

topicPublisher.publish(orderMessage);
```

Each subscriber (both the logging service and the order processing service) subscribes to this topic:

```
TopicSubscriber topicSubscriber = topicSession.createSubscriber(ordersTopic);

Message m = topicSubscriber.receive();
```

Figure 2 illustrates the publish-subscribe topology.



PROBLEMS

Follow the instructions in the document "Getting Started with Amazon's Web Services for Lessons Plans" for getting an Amazon Web Services account and setting up your system.

1. Amazon's Simple Queue Service (SQS) is a queuing system that offers many of the properties needed for an asynchronous messaging API. Adapt the queuing API discussed this document, the Java Message Service (JMS) API, to use SQS (and Amazon's SimpleDB, if needed) as the underlying message store for a JMS-like queuing API.
2. Extend your solution from problem #1 to include topics for publish and subscribe scenarios.
3. Most implementations of the JMS API use traditional, transactional databases as the underlying message store. Neither SimpleDB or SQS are transactional in the traditional sense. Amazon offers a service called EC2, the Elastic Compute Cloud. These are hosted virtual images of computing instances, usually running Linux. You can install a database on these EC2 instances, such as mySQL. Create an EC2 instance (see the "Getting Started" document for more details) and port your code to run inside of this instance using a traditional RDBMS to store message metadata or the entire messages as well.

   Write up an analysis of your two competing implementations. Which one was easier to develop? Which one is more scalable? What are the weaknesses and strengths of each?