PRIORITY QUEUES WITH SIMPLE QUEUE SERVICE

Amazon's Simple Queue Service (SQS) is a distributed queuing system. SQS is similar to the straightforward queues, but with some significant differences. Like most distributed systems, SQS has to deal with consistency across datacenters.

A traditional queue is a collection of items that follows the First-In-First-Out (FIFO) structure. Traditional queues are created in Java using class constructors:

```
Queue<String> queue = new Queue<String>();
```

In traditional queues, items are added to the queue with an Enqueue method and removed from the queue with a Dequeue method.

```
queue.offer("The First message");

queue.offer("A Second message");

String message = queue.poll(); //message will be "The First message"
```

A **Priority Queue** is a specialization of the queue data structure designed for delivering messages in a different order. Messages are delivered based on priority, not time.

Priority, as defined by priority queues, is not standard. Priority can be defined per queue. With Java's **PriorityQueue** interface, priority is based on comparisons between the messages, using a **Comparator** that is passed in when the queue is constructed. If no Comparator is passed, then the *natural comparison* operator is used.

```
PriorityQueue<String> priorityQueue = new PriorityQueue<String>();
```

Messages are added and received in the same manner.

```
priorityQueue.offer("The First message");

priorityQueue.offer("A Second message");

String message = priorityQueue.poll(); //message will be "A Second message"
```

Notice that the message received first is the second one added. This is because "A Second message" is first alphabetically. The String datatype's natural comparison is (essentially) alphabetical comparison.

To create an explicit comparison, a Comparator can be created. The best way to do this is to wrap the String inside a custom class.

```
public class StringMessage {

    public StringMessage(){
    }

    public StringMessage(String message, int priority){
        this.message = message;
        this.priority = priority;
    }

    private String message;
    private int priority;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public int getPriority() {
        return priority;
    }

    public void setPriority(int priority) {
        this.priority = priority;
    }
}
```

Next, the priority queue is created with a custom Comparator.

```
PriorityQueue<StringMessage> priorityQueue = new PriorityQueue<StringMessage>
                (2, new Comparator<StringMessage>(){
                    public int compare(StringMessage a, StringMessage b){
                        if(a.getPriority() > b.getPriority())
                            return 1;
                        else if(a.getPriority() < b.getPriority() )
                            return -1;
                        else
                            return 0;
                    }
            });
```

Finally, we create **StringMessage** objects and send them to the queue. Notice that explicit priorities are used.

```
StringMessage msg1 = new StringMessage("The First Message", 10);
StringMessage msg2 = new StringMessage("Second Msg", 5);

priorityQueue.offer(msg1);
priorityQueue.offer(msg2);

System.out.println(priorityQueue.poll().getMessage()); //displays “Second Msg”
```

A queue with SQS is created using a **CreateQueue** message. The queue can be accessed from any platform using any language capable of REST or SOAP. After a queue is created, messages are queued by **sending** a message to the queue.

```
AmazonSQS service = new AmazonSQSClient(accessKeyId, secretAccessKey);
CreateQueue request = new CreateQueue();
request.setQueueName(queueName);
CreateQueueResponse response = service.createQueue(request);
SendMessage sendMessageRequest = new SendMessage();
sendRequest.setQueueName(queueName);
```

```
sendRequest.setMessageBody("Hello, world");

SendMessageResponse sendResponse = service.sendMessage(sendRequest);
```

To dequeue a message, you **receive** a message from the queue and then **delete** it.

```
ReceiveMessage request = new ReceiveMessage();

request.setMaxNumberOfMessages(1);

request.setQueueName(queueName);

ReceiveMessageResponse response = service.receiveMessage(request);

if (response.isSetReceiveMessageResult()) {

     ReceiveMessageResult  receiveResult = response.getReceiveMessageResult();

            List<Message> messagesList = receiveResult.getMessage();

            for (Message message : messagesList) {

                DeleteMessage delRequest = new DeleteMessage();

                delRequest.setQueueName(queueName);

                delRequest.setReceiptHandle(message.getReceiptHandle());

                service.deleteMessage(delRequest);

            }

}
```

Many distributed queuing implementations are transactional. SQS queues have a **visibility** timeout. If the received message is not deleted by the end of the timeout, the message is visible to other clients. During this timeout, no other client will be able to receive the message.

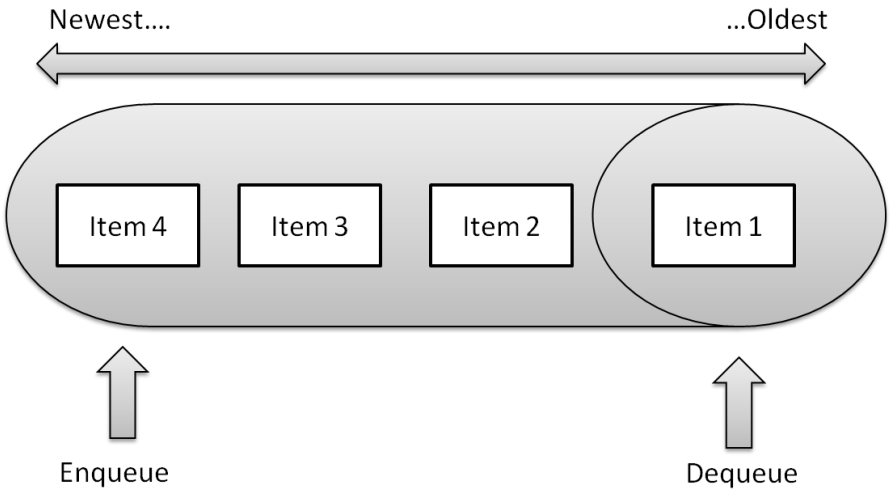

FIGURE 1 – HOW TRADITIONAL QUEUES WORK

Because SQS queues are distributed, not all systems within Amazon hold the same messages. Further, using the principal of **eventual consistency**, it is possible that messages will not be received in the same order as they are sent in all cases.

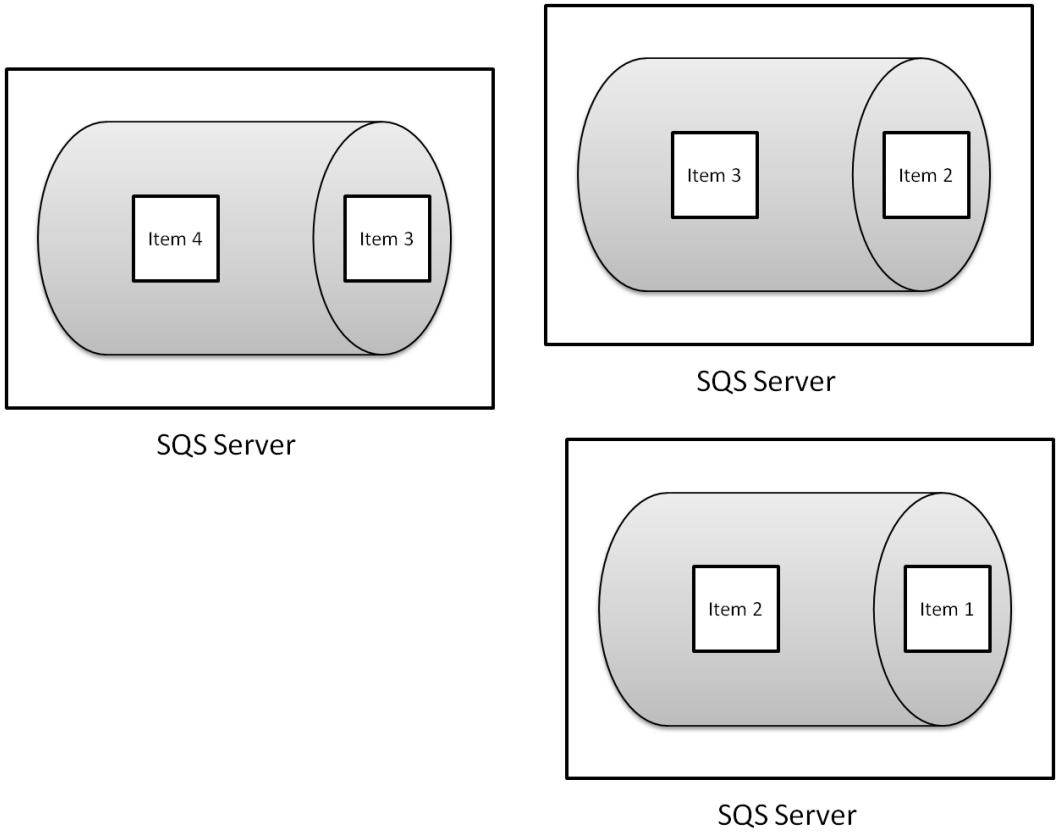

FIGURE 2 – SQS PHYSICAL TOPOLOGY

Keep in mind that SQS queues are neither FIFO nor based on any *a priori* priority data. Messages are assumed to be unordered.

The SQS reference documentation at http://docs.amazonwebservices.com/AWSSimpleQueueService/2008-01-01/SQSDeveloperGuide/ contains more details of SQS use and syntax.

PROBLEMS

The following problems each use a sample data set generated by the attached Java program "InstallSQS.java". 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. Java's standard interface for priority queues is java.util.PriorityQueue<E>. Create a class that implements this interface, wrapping SQS.

2. java.util.BlockingPriorityQueue<E> is an extension of the queue interface that adds additional capabilities, including blocking. Extend your code to implement BlockingPriorityQueue<E>.
3. Bonus question: How successful do you feel your implementations were? Would you recommend using a priority queue ver using Amazon's SQS API? What advantages or disadvantages does your implementation offer? And do they comply with the java.util.PriorityQueue<E> and java.util.BlockingPriorityQueue<E> specs completely?