Assignment 7 - Solution

Problem 1

Consider a system that uses persistent queues for requests and replies, but where the **queues do not support transactions**.

Each queue simply supports the **atomic operations:**

- Enqueue(r)
- Dequeue(r)
- ReadTop(r) - non-destructive read of first element in queue.

Suppose the system adopts the three-transaction request-processing model

- Txn1 enqueues a request,
- Txn2 executes a request and enqueues a reply,
- Txn3 dequeues a reply.

Problem 1A

Suggest an implementation of Txn2 such that each request executes **at least once.**

1. Call **ReadTop(r)** to read request at top of queue
2. Run a transaction to process r.
3. After processing r, **Dequeue it.**

If the system stays up long enough, the top request will be processed at least once. If there's a failure after the ReadTop and before the Dequeue, then after recovery the request will be processed a second time giving at-least-once semantics.

Problem 1B
Suggest another implementation of Txn2 such that each request executes **at most once.**

1. **Deque(r)**
2. Run a transaction to process r.

If there's a failure that causes the latter transaction to abort, then r may not be processed. But since it's dequeued only once, it will be processed at most once.

Problem 1C

Suggest another implementation of Txn2 such that each request executes **exactly once**, under the assumption that each transaction executed by Txn2 is **testable**.

```
Call **ReadTop(r).**
IF r has already executed
        (we can find out because the
        transaction executed for r is testable),
THEN
        **dequeue(r),**
ELSE
        run a transaction to process r.
        After transaction commits, **dequeue(r).**
```

Problem 2

Consider a system that uses **persistent queues that do not support transactions**.

Each queue supports the **atomic operations :**

- Enqueue(r, qn) – enqueue r to end of queue qn,
- Dequeue(r, qn) – dequeue top from queue qn & return it in r,
- ReadTop(r, qn) - a non-destructive read of the first element in qn.

Suppose there is one single-threaded server that processes requests from the request queue.

```
While (true) do {
        Write(Last = 0);
        // 0 is a value that is different from any request

        StartTransaction;
                ReadTop(r, RequestQueue);
                s = ProcessRequest(r);
                Write(Last = r);
                Write(Reply = s);
        Commit;

        Enqueue(Reply, ReplyQueue);
        Dequeue(x, RequestQueue);
}
```

In the above server program, Last and Reply are transactional data items in stable storage.

Write operations on them are undone if the transaction that invokes them aborts.

If the server fails, the values of local variables r, s, and x are lost.

After the server recovers, before restarting execution of the above program it runs the following recovery program:

```
ReadTop(r, RequestQueue)
IF Last = r
THEN {
        Dequeue(x, RequestQueue);
        Enqueue(Reply, ReplyQueue)
}
```

Although the server fails occasionally, it is highly available.

Assume that **ProcessRequest() supports transactions**.
If its transaction aborts, it is undone as if it was never called.

Problem 2A.
The above programs ensure that for each request r, it is processed **exactly once**, even in the presence of server failures (assuming no other kinds of failures).

True.
If the containing transaction commits, ProcessRequest() is executed and its reply is durable.

Recovery code ensures that it is not executed again.

If the containing transaction aborts, ProcessRequest() is undone.

Problem 2B.
The above programs ensure that each reply is enqueued at least once, even in the presence of server failures (assuming no other kinds of failures).

False.

If the server fails during recovery after it dequeues x and before it enqueues reply, it will never enqueue a reply.

Replies may be lost.

Problem 2C.
Atomic operation **EnqueueDequeue(r, qn1, s, qn2)** which enqueues r on qn1 and dequeues top of qn2 and returns it in s.
In the server program, replace the last two lines by EnqueueDequeue(s, ReplyQueue, x, RequestQueue).

In the recovery program, replace the then-clause by EnqueueDequeue(Reply, ReplyQueue, x, RequestQueue).

Each **reply is enqueued exactly once**, even in the presence of server failures (assuming no other kinds of failures).

True. Adding a reply is coupled with removing the corresponding request in an atomic operation.
Exactly one reply is enqueued when the request dequeued.

Problem 3.
ProcessRequest() has side effects, and does not execute atomically.

Problem 3A.
The above programs ensure that for each request r, it is processed **exactly once**, even in the presence of server failures (assuming no other kinds of failures).

False.
If the transaction containing ProcessRequest() aborts after the call (for example we run out of disk space or hit a software bug), and then the server crashes, then ProcessRequest() will be called again after recovery.

Problem 3B.
The above programs ensure that each **reply is enqueued at least once**, even in the presence of server failures (assuming no other kinds of failures).

Same as problem 2B.

False.

If the server fails during recovery after it dequeues x and before it enqueues reply, it will never enqueue a reply.

Replies may be lost.

Problem 3C.

Atomic operation **EnqueueDequeue(r, qn1, s, qn2)**.

In the server program,

EnqueueDequeue(s, ReplyQueue, x, RequestQueue).

In the recovery program, replace the then-clause by EnqueueDequeue(Reply, ReplyQueue, x, RequestQueue).

Each **reply is enqueued exactly once**, even in the presence of server failures (assuming no other kinds of failures).

True.

Adding a reply is coupled with removing the corresponding request in an atomic operation. Failures may cause a request to be executed several times in ProcessRequest(r) generating multiple replies. But exactly one reply is enqueued when the request is dequeued.