6.033 Computer System Engineering: Spring, 1999

Quiz 3

Most problems on this exam are multiple-choice questions. In order to receive credit you must fill in the blank(s) or mark the correct answer or answers for each question.

Put your name on this cover sheet AND at the bottom of each page of this booklet.

Some problems may be much harder than others. Read them all through first and attack them in the order that allows you to make the most progress. Be neat. If we can't figure out your answer we can't give you credit.

THIS IS AN OPEN BOOK, OPEN NOTES QUIZ.

Circle your recitation section number:

10:00	1. Sussman/Rahul	11. Rudolph/Forbess	
11:00	2. Sussman/Doering	4. Seshan/Boyapati	6. Rudolph/Rahul
12:00	5. Balakrishnan/Fu		
1:00	3. Seshan/Forbess	7. Saltzer/Hu	10. Teller/Boyapati
	8. Balakrishnan/Shlyakhter		
2:00	9. Saltzer/Shlyakhter	12. Teller/Doering	

Do not write in the boxes below

1-4 (XX/30)	5-14 (XX/54)	15-16 (XX/16)	Total (XX/100)

Name: ______________________________

A. Any of this sound familiar?

1. [6 points]: Failures are

A. Faults that are latent.

B. Errors that are contained within a module.

C. Errors that propagate out of a module.

D. Faults that turn into errors.

(circle letter to left of best answer)

2. [8 points]: What protocol(s) does system R use?

A. A logging protocol that allows undo of pending transactions that were started before the most recent checkpoint;

B. A logging protocol that allows redo of transactions that were committed after the most recent checkpoint;

C. An at-most-once protocol for transactions that update state on remote computers;

D. An end-to-end protocol to authenticate users.

(circle letter to left of ALL THAT APPLY)

3. [8 points]: What are the observations that motivated LFS?

A. RAM is cheap and therefore operating systems can maintain a large buffer cache.

B. The buffer cache absorbs most disk reads.

C. Disk writes must eventually go through the disk for durability.

D. Disk seeks are expensive (in time).

(circle letter to left of ALL THAT APPLY)

4. [8 points]: What assumptions does Coda make with respect to concurrent updates to different replicas of the same file?

A. It assumes that these conflicts seldom happen.

B. It assumes that these conflicts can be automatically detected.

C. It assumes that all conflicts can be automatically resolved later.

D. It assumes that these conflicts cannot happen.

(circle letter to left of ALL THAT APPLY)

B. AtomicPigeon!.com

After selling PigeonExpress!.com and taking a trip around the world, Ben Bitdittle is planning his next startup, AtomicPigeon!.com. AtomicPigeon improves over PigeonExpress by offering an atomic data delivery system.

Remember that when transmitting a pigeon, Ben's software prints out a little header and writes a CD, both of which are given to the pigeon. The header contains the GPS coordinates of the sender and receiver, a type (request or acknowledgement), and a sequence number:

```
struct header {
   GPS dst;   // GPS coordinates of destination
   GPS src;   // GPS coordinates of source
   int type;  // REQUEST or ACK
   int seq;   // sequence number
}
```

Ben starts with the code for the simple end-to-end protocol (BEEP) for PigeonExpress!.com. He makes a number of modifications to the sending and receiving code. At the sender, Ben simplifies the code. The BEEP protocol transfers only a single CD:

```
int sequence;// global sequence number

init_sender() { sequence = 0; }

BEEP(GPS dest, CDtype CD) {
   struct header h;

   h.src = my_GPS;// set src to my GPS coordinates
   h.dst = dest; // set dst to destination
   h.type = REQUEST; // set type to REQUEST
   h.seq = sequence;
   // loop until we receive the corresponding ack, retransmitting
   // if needed
   while (h.seq == sequence) {
      send pigeon with h, CD;// print header, write CD, and transmit
      wait 2,000 seconds;
   }
}
process_ack(struct header h) {
   if (h.seq == sequence) {
      sequence = sequence + 1;
   }
}
```

Pending and incoming acknowledgements are processed only when the sender is waiting.

Ben makes a small change to the code running on the receiving computer. He adds a variable `nextSequence` at the receiver, which is used by `process_request` to filter duplicates:

```
int nextSequence; // the next expected sequence number

init_receiver() { nextSequence = 0; }

process_request(struct header h, CDtype CD) {
  if (h.seq == nextSequence) { // the expected seq #?
       process(nextSequence, h.src, CD); // yes, process data
       // increase next expected sequence number
       nextSequence = nextSequence + 1;
  }
  h.dst = h.src; // destination is where the request came from
  h.src = my_GPS; // source is my GPS coordinates
  h.seq = h.seq; // seq of ACK is seq of request
  h.type = ACK;
  send pigeon with h;
}
```

`init_sender` and `init_receiver` are called once when the sending and receiving computer, respectively, start.

The assumptions for the pigeon network are the same as in quiz 2:

1. Some pigeons might get lost, but, if they arrive, they deliver data correctly (uncorrupted)
2. The network has one sender and one receiver
3. The sender and the receiver are single-threaded

5. [6 points]: Assume the sender and receiver do not fail (i.e., the only failures are that some pigeons may get lost). Does `process` in `process_request` process the value of CD exactly once?

A. Yes, since `sequence` is a nonce and the receiver processes data only when it sees a new nonce.

B. No, since `sequence` and the `nextSequence` number may get out of sync, because the receiver acknowledges requests even when it skips processing.

C. No, since if the acknowledgement isn't received within 2,000 seconds, the sender will send the same data again.

D. Yes, since pigeons with the same data are never retransmitted.

(circle letter to left of best answer)

Ben's new goal is to provide atomicity, even in the presence of sender or receiver failures. The reason Ben is interested in providing atomicity is that he wants to use the pigeon network to provide P-commerce (analogous to E-commerce). He would like to write applications of the form:

```
transfer(int amount, GPS dst) {
   CDtype CD;

   write(amount, CD); // write amount on a CD
   BEEP(dst, CD);    // send amount
}
```

The amount always fits on a single CD.

You may assume that if the sender or receiver fails, the failure is fail-fast. For now, assume that if the sender or receiver fails, it just stops and does ***not*** reboot; later, we will relax this constraint.

6. [6 points]: Given the current implementation of the BEEP protocol and assuming that only the sender may fail, what could happen during, say, the 100th call to `transfer`?

A. That transfer might never succeed.

B. That transfer might succeed.

C. `process` in `process_request` might process `amount` more than once.

D. `process` in `process_request` might process `amount` exactly once.

(circle letter to left of ALL THAT APPLY)

Ben's goal is to make transfer always succeed by allowing the sender to reboot and finish failed transfers. **So, for the rest of this quiz assume that after the sender fails, it reboots, calls `init_sender`, and restarts the application. The application first runs a recovery procedure, which retries a failed transfer, if any.**

To allow for restartable transfers, Ben supplies the sender and the receiver with stable storage that never fails. On the stable storage, Ben stores a log, in which each entry has the following form:

```
struct log_entry {
   int type;        // BEGIN or COMMIT
   int seq;         // a sequence number
}
```

The main objective of the sender's log is to allow `init_sender` to restore the value of `sequence` and to allow the application to restart an unfinished transfer, if any.

Ben edits `transfer` to use the log:

```
transfer(int amount, GPS dst) {
   CDtype CD;

   write(amount, CD);          // write amount on a CD
   add_log(BEGIN, sequence);   // append BEGIN record
   BEEP(dst, CD);              // BEEP increases sequence
   add_log(COMMIT, sequence - 1); // append COMMIT record
}
```

`add_log` atomically appends a record to the log on stable storage. If `add_log` returns, the entry has been appended. You may assume that logs contain sufficient space for new records and they don't have to be garbage collected.

7. [3 points]: Circle the line in this new version of `transfer` that is the commit point.

8. [3 points]: How can the sender discover that a failure caused the transfer not to complete?

A. The log contains a BEGIN record with no corresponding COMMIT record.

B. The log contains a BEGIN record with a corresponding ABORT record.

C. The log contains a BEGIN record with a corresponding COMMIT record.

D. The log contains a COMMIT record with no corresponding BEGIN record.

(circle letter to left of best answer)

Ben tries to modify `init_sender` to recover `sequence`, but his editor crashes before committing the final edit changes: the expression in the `if`-statement is missing, as indicated by a "**?**" in the code below. Your job is to edit the code such that the correct expression is evaluated.

```
init_sender() {
   struct log_entry entry;

   sequence = 0;
   go_to_last_log_entry();
   while ( ! beginning_of_log()){
      entry = read_current_entry_of_log();
      if ( ? ) { /* 1 */
         sequence = entry.seq + 1;
         break; // terminate scan of log
      }
      go_to_previous_log_entry();
   }
}
```

9. [6 points]: After you edit `init_sender` for Ben, which of the following sequences could appear in the log? (The log records are represented as <type seq>.

A. ..., <BEGIN 1>, <COMMIT 1>, <BEGIN 2>, <COMMIT 2>

B. ..., <BEGIN 1>, <BEGIN 1>, <COMMIT 1>

C. ..., <BEGIN 1>, <BEGIN 1>, <BEGIN 2>, <COMMIT 1>, <BEGIN 2>

D. ..., <BEGIN 1>, <COMMIT 1>, <COMMIT 1>

(circle letter to left of ALL THAT APPLY)

10. [6 points]: What expression should be evaluated at **?** on the line marked **1** in the `init_sender` code above?

A. `entry.type == COMMIT`

B. `entry.type == BEGIN`

C. `entry.type == ABORT`

D. `FALSE`

(circle letter to left of best answer)

11. [6 points]: Given the current implementation of the BEEP protocol what could happen, say, during the 100th call to `transfer`? (Remember only the sending computer may fail.)

A. If the sending computer keeps failing during recovery, that transfer might never succeed.

B. That transfer might succeed.

C. `process` in `process_request` might process `amount` more than once.

D. `process` in `process_request` might process `amount` exactly once.

(circle letter to left of ALL THAT APPLY)

Ben's next goal is to make `process_request` recoverable. **From now on, assume that whenever the receiving computer fails, it reboots, calls `init_receiver`, and after `init_receiver` is finished, it waits for messages and calls `process_request` on each message.**

Name: ____________________________

To make `nextSequence` recoverable, Ben tries to make similar changes to the receiver as he did to the sender. Again, his editor didn't commit all the changes in time. The missing code is marked by "**?**" and "**#**". The missing expression marked by "**?**" evaluates the same expressions as in the `init_sender` on page 6. The new missing expressions are marked by "**#**" (see the lines marked **2.** and **3**. in `process_request`):

```
init_receiver() {
   struct log_entry entry;

   nextSequence = 0;
   go_to_last_log_entry();
   while ( ! beginning_of_log()){
      entry = read_current_entry_of_log();
      if ( ? ) { /* 1. (the expression circled in question 10) */
         nextSequence = entry.seq + 1;
         break; // terminate scan of log
      }
      go_to_previous_log_entry();
   }
}

process_request(struct header h, CDtype CD) {
   if (h.seq == nextSequence) {
      add_log(#, #); /* 2. */
      process(nextSequence, h.src, CD);
      nextSequence = nextSequence + 1;
      add_log(#, #); /* 3. */
   }
   h.dst = h.src;
   h.src = my_GPS;
   h.seq = h.seq;
   h.type = ACK;
   send pigeon with h;
}
```

As you can see from the code, Ben chose not to implement a write-ahead protocol, because `process` is implemented by a third party, for example, a bank: `process` might be a call into the bank's transactional database system.

Name: ______________________________

12. [6 points]: Complete the `add_log` calls on the lines marked **2.** and **3.** in `process_request` such that `nextSequence` will be recoverable. (Fill in the empty spaces.)

```
add_log(                    ,                    ); /* 2. */
add_log(                    ,                    ); /* 3. */
```

13. [6 points]: Can `process` in `process_request` be called multiple times for a particular call to `transfer`?

A. No, because `nextSequence` is recovered and `h.seq` is checked against it.

B. Yes, because failed transfers will be restarted and result in the acknowledgement being retransmitted.

C. Yes, because after 2,000 seconds a request will be retransmitted.

D. Yes, because the receiver may fail after `process`, but before it commits.

(circle letter to left of best answer)

14. [6 points]: How should `process`, called by `process_request`, be implemented to guarantee exactly-once semantics for transfers? (Remember that the sender is persistent.)

A. As a normal procedure call;

B. As a remote procedure call;

C. As a nested transaction;

D. As a top-level transaction.

(circle letter to left of best answer)

C. Almost done with 6.033!

A system that attempts to make operations atomic writes the following type of records to a log maintained on stable storage:

- `<BEGIN i>`: operation i starts.
- `<i, x, old, new>`: operation `i` writes the value `new` over the value `old` for the variable `x`.
- `<COMMIT i>`: operation `i` commits.
- `<ABORT i>`: operation `i` aborts.
- `<CHECKPOINT i, j, ...>`: At this checkpoint, operations `i, j, k`, ... are pending.

Operations start in numerical order. The recovery procedure is using a backout-list log protocol. A crash occurs, and the recovery procedure finds the following log records:

```
    <CHECKPOINT 17, 51, 52>
    <BEGIN 53>
    <BEGIN 54>
    <53, y, 5, 6>
    <53, x, 5, 9>
    <COMMIT 53>
    <54, y, 6, 4>
    <BEGIN 55>
    <55, z, 3, 4>
    <ABORT 17>
    <51, q, 1, 9>
    <BEGIN 56>
    <55, y, 4, 3>
    <COMMIT 54>
    <55, y, 3, 7>
    <COMMIT 51>
    <BEGIN 57>
    <56, x, 9, 2>
    <56, w, 0, 1>
    <COMMIT 56>
    <57, u, 2, 1>
/***************** crash happened here *************/
```

15. [8 points]: Which operations mentioned in the this part of the log are losers (i.e., whose changes have to be undone)?

A. 17
B. 51
C. 52
D. 53
E. 54
F. 55
G. 56
H. 57

(circle letter to left of ALL THAT APPLY)

16. [8 points]: Given this part of the log, does the system provide isolation of operations?

A. Yes, all of these operations are serializable.
B. No, operation 55 is using a value of `y` that operation 54 has changed, but not yet committed.
C. We don't know, since the system crashed.
D. No, because operation 53 committed before operation 17 aborted.

(circle letter to left of best answer)

END OF QUIZ 3

(Enjoy your summer!)