III Not quite Replicated State Machines

Ben has come up with a simple solution to lab 2B. The pseudocode for his solution is shown on the next page. This pseudocode captures the following strategy:

- The client sends an RPC tagged with its client ID and a unique sequence number to the primary;
- The primary forwards the request to the backup;
- The backup performs the operation (unless it was already performed, in which case it looks up the result from its duplicate detection table), updates its RPC duplicate detection table, and sends back to the primary both the *result* of the operation (possibly from the duplicate detection table) *and* the current (new) value of the corresponding key (not from the duplicate detection table);
- The primary updates its key/value store with the value provided by the backup (instead of re-executing the operation), updates its duplicate-detection table, and sends the response provided by the backup to the client;
- If the backup told the primary it couldn't perform the operation (e.g., because it is in a view change), then the primary tells the client to retry.

Ben points out to Alyssa that his design has only *one* retry loop: namely at the client. In his implementation, there is no retry loop at the primary (or backup); if the primary times out waiting for a response from the backup, it returns an error to the client, and it is the client's job to retry (with the same client ID and sequence number). Furthermore, he argues that his implementation is correct, because it has a well-defined commit point for each operation (e.g., when the backup executes an operation, and updates its key/value table and duplicate-detection table). Alyssa, however, is suspicious about Ben's implementation.

Ben’s pseudocode for lab 2B:

```
client_puthash(k, v):
  seq = choose sequence id
  while True:
    send (puthash, client_id, seq, k, v) to primary
    if success: return result
    else ask view server for primary

primary_puthash(client_id, seq, k, v):
  hold global lock
  if (client_id, seq) in dups:
    return dups[(client_id, seq)]
  if there is a backup:
    send (backup_puthash, cur_viewnum, client_id, seq, k, v) to backup
    if error: return error
    get (result, newval) from backup’s reply
    db[k] = newval
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
  dups[(client_id, seq)] = result
  return result

backup_puthash(arg_viewnum, client_id, seq, k, v):
  hold global lock
  if arg_viewnum != cur_viewnum: return error
  if (client_id, seq) in dups:
    result = dups[(client_id, seq)]
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
    dups[(client_id, seq)] = result
  return (result, db[k])
```

Ben’s pseudocode for lab 2B, repeated for your convenience (so you don’t have to flip pages):

```
client_puthash(k, v):
  seq = choose sequence id
  while True:
    send (puthash, client_id, seq, k, v) to primary
    if success: return result
    else ask view server for primary

primary_puthash(client_id, seq, k, v):
  hold global lock
  if (client_id, seq) in dups:
    return dups[(client_id, seq)]
  if there is a backup:
    send (backup_puthash, cur_viewnum, client_id, seq, k, v) to backup
    if error: return error
    get (result, newval) from backup’s reply
    db[k] = newval
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
  dups[(client_id, seq)] = result
  return result

backup_puthash(arg_viewnum, client_id, seq, k, v):
  hold global lock
  if arg_viewnum != cur_viewnum: return error
  if (client_id, seq) in dups:
    result = dups[(client_id, seq)]
  else:
    result = db[k]           // result is prev value
    db[k] = hash(db[k] + v)  // do the puthash
    dups[(client_id, seq)] = result
  return (result, db[k])
```

3. [15 points]: In the presence of failures, Ben's implementation doesn't guarantee at-most-once RPC. Show a timing diagram with RPC requests and responses that demonstrates incorrect behavior for `PutHash` operations. For each request sent to the primary, clearly indicate which client sent it, and what the sequence number was. Clearly indicate any messages lost between the primary and the backup, and any primary or backup failures.

```
Primary                                  Backup
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
   |                                        |
```

Answer:

See the diagram below for a sequence of events that causes a PutHash to be executed twice.