Homework 3

CS425/ECE428 Spring 2019

Due: Monday, April 8 at 12:00 **NOON**

NO LATE SUBMISSIONS ACCEPTED

1. Consider a Chord system with 12-bit identifiers. It has the following nodes, listed in hex and decimal for your convenience:

014, 017, 066, 06b,	20, 23, 102, 107
095, 0a6, 0a9, 0c0,	149, 166, 169, 192
0dd, 105, 147, 153,	221, 261, 327, 339
15e, 175, 17f, 1dc,	350, 373, 383, 476
1f2, 21e, 27f, 2de,	498, 542, 639, 734
353, 3a4, 3bf, 3ce,	851, 932, 959, 974
403, 416, 442, 456,	1027, 1046, 1090, 1110
45c, 464, 483, 4aa,	1116, 1124, 1155, 1194
4ca, 4e8, 522, 539,	1226, 1256, 1314, 1337
55e, 571, 60d, 658,	1374, 1393, 1549, 1624
67b, 689, 6d3, 6f1,	1659, 1673, 1747, 1777
6fb, 712, 738, 741,	1787, 1810, 1848, 1857
749, 74f, 7d1, 7e9,	1865, 1871, 2001, 2025
849, 865, 8f2, 91b,	2121, 2149, 2290, 2331
926, 984, 996, 9b8,	2342, 2436, 2454, 2488
9b9, 9c1, a21, a43,	2489, 2497, 2593, 2627
a5c, a62, a6a, a72,	2652, 2658, 2666, 2674
a92, aac, ac1, ada,	2706, 2732, 2753, 2778
b01, b17, b21, b78,	2817, 2839, 2849, 2936
bb0, bcc, bd9, bdc,	2992, 3020, 3033, 3036
bf7, c18, c1b, c98,	3063, 3096, 3099, 3224
caf, cc0, ce5, d13,	3247, 3264, 3301, 3347
d9c, e0c, e27, e49,	3484, 3596, 3623, 3657
e63, e6d, e7a, edc,	3683, 3693, 3706, 3804
f4a, f5d, f71, fd6,	3914, 3933, 3953, 4054

(a) (4 points) List the finger table of node 0x926 (2342)

Solution:

i	*ft[i]*
0	2436 (984)
1	2436 (984)
2	2436 (984)
3	2436 (984)
4	2436 (984)
5	2436 (984)
6	2436 (984)
7	2488 (9b8)
8	2627 (a43)
9	2936 (b78)
10	3484 (d9c)
11	327 (147)

(b) (4 points) List the nodes that 0x926 (2342) would contact during a lookup of the key 0x123 (291)

Solution:

2342	(926)
3484	(d9c)
4054	(fd6)
221	(0dd)
261	(105)
327	(147)

(c) (4 points) Identify the nodes that will store the largest expected number of keys and the smallest. (Assume for now that a key is stored at only the successor node.) What is the ratio of their expected storage?

Solution: 1549 (60d) will store the largest expected number of keys. (It will store 156/4096 of all keys.) 2489 (9b9) will store the smallest number of keys. (1/4096) Ratio: 156/1

(d) (4 points) A power outage takes out half the nodes: the ones with even identifiers. Assume no stabilization algorithm has had a chance to run, and so the finger tables have not been updated. List the nodes that 0x7e9 (2025) would contact to look up the key 0x480 (1152). (When a node in the normal lookup protocol tries to contact a finger entry that is no longer alive, it switches to the next best finger that is alive.)

Solution: 0x7e9 would normally contact 0x14 (its largest finger) but that failed, so it contacts 0xbf7 instead. 0bxf7 then contacts 0x403 (its largest finger). All nodes between 0x403 and 0x480 are even, so 0x403 relies on its successor lists to contact 0x483, which is the target of the lookup.

2. (a) (8 points) Use an RPC compiler, such as Apache Thrift, to answer this question. Write down an interface specification for a reader/writer locking service. Your API should allow you to create a new lock and then lock/unlock it for reading and writing.
Use the RPC compiler to generate an implementation of your protocol. Include in your submission the code for your interface definition (with comments), and a page each of the generated stub and skeleton files.

Solution: A sample solution in Apache Thrift.

rwlock.thrift

```
1  namespace cpp rwlock
2
3  struct rwlock {
4      1: i32 no,
5  }
6
7  service ReadWriteLockService {
8      // create the lock
9      rwlock initLock(),
10
11     // acquire the read lock
12     void RLock(1:rwlock l),
13
14     // release the read lock
15     void RUnlock(1:rwlock l),
```

```
16
17      // acquire the write lock
18      void Lock(1:rwlock l),
19
20      // release the write lock
21      void Unlock(1:rwlock l)
22  }
```

To get the generated files (in C++), use

```
1  thrift -r --gen cpp rwlock.thrift
```

Another sample solution in gRPC.

rwlock.proto

```
1  syntax = "proto3";
2
3  service ReadWriteLockService {
4    rpc initLock (void) returns (rwlock) {}
5    rpc RLock (rwlock) returns (void) {}
6    rpc RUnlock (rwlock) returns (void) {}
7    rpc Lock (rwlock) returns (void) {}
8    rpc Unlock (rwlock) returns (void) {}
9  }
10
11 message rwlock {
12   int32 no = 1;
13 }
14
15 message void {}
```

To get the generated files (in Python), use

```
1  python -m grpc_tools.protoc -I. --python_out=. \
2         --grpc_python_out=. rwlock.proto
```

(b) (2 points) Identify a function in the C, Go, Python, or Java standard library that has a side effect but is idempotent. Briefly explain your answer.

Solution:
`int fflush(FILE* stream)` in C stdio.h header.
`func ToUpper(string []byte) []byte` in Golang bytes pkg.
`void HashMap.clear()` in Java java.util.HashMap package.
`int(x)` in Python build-in.

3. (a) (3 points) Consider a Raft cluster with five nodes, with logs as described by follows. Each event in a log is denoted by a letter; different letters represent different events, and the subscript indicates the term of the event.

- S_1: committed: a_1, b_2, c_2, uncommitted: d_3
- S_2: committed: a_1, b_2, uncommitted: c_2, d_3, e_6

- S_3: committed: a_1, b_2, uncommitted: c_2
- S_4: committed: a_1, b_2, uncommitted: c_2, d_3, e_6, f_6
- S_5: committed: a_1, b_2, c_2, uncommitted: g_5, h_5, i_5, j_5, k_5

Which of the five nodes could be elected leader? Explain.

> **Solution:** Here are the nodes ordered in terms of the up-to-date check: S_3, S_1, S_5, S_2, S_4. So S_5, S_2, or S_4 could get elected leader, the rest cannot get a majority of the votes.

(b) (2 points) Is d_3 guaranteed to be eventually committed? Explain; you may need to offer a sequence of events.

> **Solution:** No; if S_5 gets elected leader, it will overwrite the uncommitted part of the logs of the followers and d_3 will not be used.

(c) (3 points) Not related to the previous question, describe a sequence of events where three nodes (out of five) could be in the leader state.

> **Solution:** Consider five nodes S_1, S_2, S_3, S_4 and S_5 with the following set of events:
>
> 1. S_1 gets elected with 5 votes as the Leader.
> 2. S_1 gets partioned out of the network.
> 3. S_2 gets elected with 4 remaining votes as the Leader.
> 4. S_2 gets partioned out of the network.
> 5. S_3 gets elected with 3 remaining votes (still majority) as the Leader.
>
> At the end, S_1, S_2 and S_3 will be in Leader state.

4. (a) (4 points) Consider the following transaction (T1):

```
1: x = a.getbalance()
2: y = b.getbalance()
3: c.withdraw(x-y)
4: a.deposit(x-y)
```

List when the locks on each of the objects are acquired or upgraded, and what type of lock is acquired. (`x` and `y` are local variables to the transaction)

> **Solution:**
>
> ```
> At 1 acquire read lock on a;
> At 2 acquire read lock on b;
> At 3 acquire write lock on c;
> At 4 upgrade read lock on a to write lock;
> ```

(b) (3 points) Consider a second transaction (T2):

```
1: z = b.getbalance()
2: w = c.getbalance()
3: c.withdraw(z-w)
4: b.deposit(z-w)
```

(c) (3 points) Show an interleaving of T1 and T2 that is serially equivalent, but impossible under two-phase locking (strict or reader/writer)

Solution:

```
T1.1: x = a.getbalance()
T1.2: y = b.getbalance()
T1.3: c.withdraw(x-y)
        T2.1: z = b.getbalance()
        T2.2: w = c.getbalance()
        T2.3: c.withdraw(z-w)
        T2.4: b.deposit(z-w)
T1.4: a.deposit(x-y)
```

Conflicts here are T1.2–T2.4, T1.3–T2.2, T1.3–T2.3. In the interleaving above these all follow the transaction order T1, T2, therefore the interleaving is serially equivalent. Hoewver, with 2PL, T1 will acquire a lock on c in T1.3 and not release it until after the commit, which would prevent this interleaving from occuring.

(d) (3 points) Show an interleaving of T1 and T2 that is impossible with strict two-phase locking but possible with non-strict locking (reader/writer)

Solution:

```
T1.1: read lock a; x = a.getbalance()
T1.2: read lock b; y = b.getbalance()
    T2.1: read lock b; z = b.getbalance()
T1.3: write lock c; c.withdraw(x-y)
T1.4: upgrade lock a; a.deposit(x-y)
T1 commit, releases locks
    T2.2: read lock c; w = c.getbalance()
    T2.3: upgrade lock c; c.withdraw(z-w)
    T2.4: upgrade lock b; b.deposit(z-w)
    T2 commit, release all locks
```

This solution would be impossible with exclusive locks since T1.2 would acquire a lock on b, which would prevent T2.1 from executing.

(e) (3 points) Suppose that instead of lock upgrades, transactions released a read lock and then acquired a write lock. Show a non-serially equivalent interleaving that would be possible in this situation.

Solution: Below is a non-serially equivalent execution

```
  T2: read lock b; z = b.getbalance()
  T2: read lock c: w = c.getbalance()
  T2: read unlock c ...
T1: read lock a; x = a.getbalance()
T1: read lock b; y = b.getbalance()
T1: write lock c; c.withdraw(x-y)
T1: read unlock a; write lock a; a.deposit(x-y)
T1: unlock all, commit
  T2:               write lock c; c.withdraw(z-w)
  T2: read unlock b; write lock b; b.deposit(z-w)
  T2: unlock all, commit
```