HW1 Solutions: CS425 FA20

1. (Graded by: Ishani Janveja) [3+4+3=10]
    a. On-demand instances are guaranteed, at a fixed price (given an instance type with CPU count and memory). Both AWS spot instances and Google Cloud preemptible instances are “best effort” and make excess (free) computing capacity available to users at a lower price than on-demand instances. Further, spot/preemptible instances may be terminated if resources are unavailable or market price goes above the bid price (AWS).
    b. The key difference is that AWS spot instances have variable pricing (dependent on a bid by the user), dependent on current market and resource utilization. Google Cloud’s preemptible instances have a fixed price. In AWS spot instances the user bids a price and gets the instance only if the spot price is below the bid price--the spot price varies (due to market and utilization) and if it exceeds the bid price, the instance is terminated. The user pays the current spot price. Other differences include: i) differences in notice before termination (30 s guaranteed in GCloud, 2 mins in AWS - these were the values at the time of writing this solution).; ii) differences in automatica restart -- GCloud preemptible instances auto-restart, while in AWS the request remains active for a persistent spot instance.
    c. Spot/preemptible instances are good choices when downtime and failure is acceptable, and low pricing is important. Some examples include testing runs (unit test, integration testing), machine learning training (not testing), etc. One would not run a stateful website or a critical financial application on a spot/preemptible instance.

2. (Graded by: Binyao Jiang)
    This solution chains two MapReduces after one another.
    // m candidates

```
Map1 (key=empty, value = vote V):
        for i from 1 to m-1
                for j from i+1 to m
                        Consider the vote pair (V.i, V.j). // V.i is a preferable to V.j
                        if V.i is lexicographically smaller than V.j // e.g., V.i = Alpha, V.j = Beta
                                Emit (key=(V.i, V.j), value=1) // left part of tuple is winner
                        else
                                Emit (key=(V.j, V.i), value=0) // right part of tuple is winner

Reduce1 (key=(A,B), value=array of {0,1} entries) // A is lexicographically smaller than B
        Count the number of 0s and number of 1s in array.
```

if more 1s than 0s, output (A,B) // A dominates B
else output (B,A) // B dominates A. Note that number of votes N is odd.

Map2 (key=A, value=B) // A or B may be lexicographically smaller
Emit (key=1, value=(A,B))
Reduce2 (key=1, value = array of (A,B) pairs) // only 1 reduce task
create int Carray[m] of candidates, init to 0
for each (A,B) pair in value
Carray[A]++
if some element Carray[A]==m-1
Emit (key=A, value="Condorcet winner!")
else
Find max count in Carray
Find set S of all candidates whose Carray counts == max
Emit (key=set S, value="No Condorcet winner, Highest Condorcet counts")
// Note: also ok if your solution emits only candidate(s) highest condorcet counts, as this would include a Condorcet winner

3. (Graded by: Binyao Jiang)
We use short hand Unique_Person_Name = name, start_time = start, end_time = end.

Remember -- You are writing code for a Map function and Reduce function (not Map task and Reduce task!). Note that a Map function will receive as input a single (key,value) pair, and output zero or more (key,value) pairs. A Reduce function receives multiple (key,value) pairs (all values for a given common key), and outputs zero or more (key,value) pairs.

MR1
- M1 reads from D1 and outputs (key=name, value=(location, start, end))
- R1 is identity // outputs (key=name, value=list of (location, start, end) intervals)

MR2
- M2 reads from D2 and outputs (key=name, value = positive)
- R2 is identity.

MR3 reads both MR1's and MR2's outputs.
- M3: // takes as input a key value pair
  - If value contains "positive", then emit (key=location, value=(positive, (start, end)))
  - If value is (location, start, end), then emit (key=location, value=(testcase, (start, end, name)))
- R3:

    - Receives for each key=location, two lists: list L1 of when positive individuals were around (value=(positive, *)) as well as list L2 of other individuals who visited the location at least once (value=(testcase, *)).
    - For each (value=(positive, *)) in L1, find all entries in L2 that overlap with (start, end) times (fancy datastructures not needed)--for such entries Emit (key=L2 entry's name, value=null). Note that there might be multiple such entries, across reduce tasks!

MR4 reads from MR3

- M4: Receives (key=name, value=null). Identity.
- R4: Receives (key=name, value=(null, null,...)) (multiple possible entries). Emit (key=name, value=needs-to-be-tested).

Note that the above solution also includes the names of known infected individuals (from D2). If you'd like to eliminate these, then you can add an additional MR.

4. (Graded by: Xin Tong) [4+3+3]
    a. Each process p is pinged by K+M-2 other processes. If p fails but at least one pinger is non-faulty, then p's failure will be detected. p's failure goes undetected only if all its pingers are also failed -- this happens only if the number of failures is >= (K-1+M-1) + 1 (the -1 and +2 exclude/include the process p itself). Thus L = K+M-1. Note that the question is not about false positives but actual crash/fail-stop failures.
    b. If K=2, L=10, from part a, M needs to be at least 9.
    c. Pinging and heartbeating protocols cannot satisfy accuracy in an asynchronous distributed system, no matter the values of R, K, M, N, L, etc.

5. (Graded by: Bhavana)
   The amount of data (suspicion-related messages) piggybacked on a given SWIM message is proportional to the i) fraction of processes that have actually failed, and ii) fraction of processes mistakenly detected as failed. Note that this max amount (for a given message) does *not* depend on the number of detecting processes, just the number of suspected/detected processes (this is because suspicion messages do not identify the originator, just the suspected node). Given this, each suspected process will add 1 entry to a SWIM piggybacked data. Since it takes O(log(N)) time to timeout a suspicion into a failure, and we are given that all suspicions result in Alive messages, each suspicion message will persist for O(log(N)) SWIM periods. Alive messages also persist for another O(log(N)) periods, per suspicion. So, in a given time unit, the last O(log(N)) time units' suspected processes still have their information spreading. Thus a typical SWIM message in steady state will piggyback information about O(log(N)) other processes.

6. (Graded by: Ruiyang)
    a. Here we calculate the maximum number of nodes that could be reached. 12 processes (excluding the sender). Note that with a TTL of k (k<=4), one would reach 4*k processes (excluding the sender).
    b. Minimum TTL = 5 (consider reaching the diametrically opposite process from the sender)
    c. TTL = 2. Sender sends query to all its neighbors (including the 21st process), and the 21st process forwards it to all the processes

7. (Graded by: Ruiyang)
    a. 2m-2. A tree with 2^m-1 nodes has m levels of nodes, and m-1 levels of edges. The highest time for a query to spread is when leaf node is the sender process. This takes m-1 hops to reach the root of the tree, and another m-1 hops to reach the farthest leaf.
    b. (2^(m-1)-2) + (2^(m-3)-1) + 1 (root)= 2^(m-1) + 2^(m-3) - 2. The sender's subtree has (2^(m-1)-2) nodes (apart from itself) and they are all reachable with a TTL of m-2. The sender's message to the tree's root reduces TTL to m-3. The "other" side of the tree, starting with the tree root, disseminates a message with TTL m-3 to (2^(m-3)-1) nodes (includes the tree root).
    c. m. The child of a root has to reach its query to a leaf on the "other" (sibling) side of the tree. It takes 1 hop to reach the tree root, and another m-1 hops to reach everyone in the other subtree (note that m hops takes care of the sender's side of the subtree).

8. (Graded by: Atul) [3+4+3]
   The ordered list is 3, 10, 14, 15, 19, 25, 30, 75.
    a. For 14, (finger table id, process id) list = (0, 15), (1, 19), (2, 19), (3, 25), (4, 30), (5, 75), (6, 3).
    b. Paths
        i. 14, 75, 3. (key 90 is at process 3)
        ii. 75, 14, 25, 30. (key 26 is at process 30).
    c. Node 30 was present in the finger tables of 10 (4th FT), 14 (4th FT), 19 (3rd FT), 25 (0th, 1st and 2nd).

9. (Graded by: Atul) [5+5]
    a. Using BitTorrent's local rarest first policy, the number of replicas for each block are A=4, B=2, C=5, D=6, E=5. So the order is: (1) B first, (2) then A, (3, 4) then C and E (tied), and (5), D.

b. Since the difference between 1st and 2nd place shards is more than 1, Pluto will not change first fetched shard, which will remain B. In other words, Pluto does not affect the Solar System.

10. (Graded by: Xin Tong) [3+3+2+2]
    a. Worst case lookup cost remains the same as Kelips, i.e., O(1) hops. Worst case is to send query to contact in file's affinity group, and then the full membership there gets the file location.
    b. Each of the 10 affinity groups has N/10 nodes and thus an O(N/10) membership list, or O(N). The memory utilization is O(N).
    c. Vs.
        i. Advantage of NoLips over Kelips:
            - A larger fraction of files will be in the sender's affinity group in NoLips (10% vs. 1/sqrt(N)), and so best case lookup cost of 0 hops (file location is present locally at sender) will be more likely in NoLips.
            - No need to rehash and redistribute nodes and files when nodes join or leave.
            - More replication of file metadata within a group.
            - Other reasonable answers are also accepted.
            - (Note: Some students answered NoLips has faster dissemination time than Kelips. This is untrue, at least in the O() world. Both are O(log(N)) dissemination time (via gossip).
        ii. Advantage of Kelips over NoLips:
            - Memory is lower (O(sqrt(N) vs. O(N)), and lookup cost is equivalent (O(1)).
            - Other reasonable answers are also accepted.