Homework 1 Solutions: CS425 FA25

Note to Students: Recommended solutions for HW1 (FA25) are below. For many questions, alternate solutions are possible and reasonable correct solutions will be accepted during grading. So please refrain from asking questions about solutions until you receive your HW grades back.

1. (Solution and Grading by: Lilia Tang)
    a. This is similar to the protocol discussed in class, with a tweak. Once infected, each subtree (left or right) takes **O(log(N/2-1) = O(log(N)))** time to disseminate the gossip within that subtree's group. After most of the sender's subtree has been infected, the expected time to infect the root of the tree is **O(1)** (it may actually be faster than that, but nothing is smaller than O(1) anyway). Then it takes the sender O(1) rounds to infect the other subtree. This is followed by another **O(log(N/2-1) = O(log(N))** to disseminate the gossip within that subtree's group. So the total time is O(log(N)) + O(1) + O(log(N)) = O(log(N)).
    b. Average load on any subtree node is O(1) since gossip is random. The root is the only one that may get a higher load. However, since each node picks the root with a small probability O(1/N), the load on the root from *each* subtree is O(1), even when the subtrees all are gossiping and pushing the message. Thus the root's load is also O(1). The answer thus is O(1).

2. (Solution by: Chirag Shetty. Grading by: Ashish Kashinath)

a. What is the key difference between AWS Lambda and AWS EC2?
Ans: Key Difference: In EC2, users manage the VMs that their program runs on. In Lambda, users only specify the program to run and resources it needs; the cloud provider manages running it on the VMs.
(Other: EC2 is priced by the number and size of VMs used. Lambda is priced by number of function calls and resource per call. Lambda functions are generally stateless; EC2 can be used to implement stateful applications)

b. What are the two key differences between AWS Lambda and AWS spot instances (think: pricing and how long instances last)?
Ans:
   i. Pricing: Lambda is priced by number of function calls and resource per call. Spot instances are priced by number and type/size of spot instance for the amount of time they are running (prices keep varying)
   ii. Lifetime: Lambda lasts for as long as the function/program is running. Spot instance runs until it is pre-empted by the cloud provider or the user shuts it down.

(Other: Key difference from (a) applies here as well since, spot instances are just EC2 instances)

c. What is the key difference between a “regular” AWS instance (EC2) and a “burstable” AWS instance?
Ans: With regular instances, the amount of *CPU* programs running on that instance can use is limited to the size of the instance chosen. With burstable instances, the programs can momentarily (for short durations) use more CPU that the size of the instance i.e “bursting”

Bonus Question (not graded): This is only allowed with CPU, not memory. Why?
Ans: CPU is a “compressible” resource - a program will run fine with lesser or more CPU, just slower or faster. Memory is not compressible - there is no way to make a program function with less than its memory requirement - so bursting is not useful (because once you give memory to a program, there is no way to take it back without killing the program)

d. What is the difference between a GPU and a “TPU” (among cloud offerings)?
Ans: GPU has a general purpose architecture that can accelerate any highly parallelizable program. TPU was specifically designed for AI/ML applications, specifically for fast matrix multiplication (using systolic arrays) . All major cloud providers have GPU offerings. Only Google has TPU offerings

Bonus Question (not graded): A major LLM model company recently incorporated TPUs into their LLM training infrastructure, along with GPUs . Which one?
Ans: OpenAI

e. What is the difference between AWS spot instances and Google Cloud preemptible instances?
Ans:
AWS spot instances prices vary dynamically (multiple times a day), while GCP preemptible instance prices are more static (changes upto once every 30 days)
Source: https://cloud.google.com/compute/docs/instances/spot
Note: GCP premptible instances and spot instances have the same pricing - premptible instances were an earlier version of spot, which could only remain running upto 24 hours. Spot instances do not have that restriction.

f. Give one example application (class) where you would prefer AWS EC2, and one where you would prefer AWS Lambda. Justify your choices briefly
Ans: Typical Applications:
1. Lambda: Event processing tasks; eg: User authentication, File/photo compression, Input parsing and writing to DB, etc
2. EC2: Any server like application that must be “always on” (think any user facing app - uber, doordash, etc)
(That said, large applications many times are often a mix of EC2 and Lambda.
General Thumbrule:
    1. *Lambda: Short running jobs with low resource requirements. Users does not want to manage VMs etc (usually developers with no devops support)*
    2. *EC2: Dynamic mix of jobs (length of jobs, resource requirements etc). User is well versed with DevOps)*

3. (Solution and Grading by: Chirag Shetty)
   Docker, Container, Virtual Machine (VM),
   Kubernetes (K8s), K8s Pod, K8s cluster. Help them please! Do the following:

   a. Define each of these 6 terms concisely (1 sentence per term).
   Ans:
   1. Containers: Containers are OS (user-space) virtualization that allow running programs in isolated environments while sharing the same host OS, each with its own files, dependencies and resource allocation.
   2. Docker: Docker is a platform to create, modify and deploy containers.
   3. VM: VM is a technology that virtualizes a physical machine to run multiple OS kernels offering full isolation between the programs running on the OSes
   4. K8s: K8s is an orchestration system used to manage containers on a cluster of physical machines or VMs
   5. K8s Pod: Pod is a container or a bundle of containers that is treated as a single unit by K8s for purposes like scheduling, creation/deletion, configuration updates, network namespace allocation etc.
   6. K8s cluster: K8s Cluster refers to a collection of physical machines or VMs managed by K8s to run pods on them.

   b. Among the four possibilities of docker, container, VM, and K8s, give one scenario where one would use each of them over the three possibilities.
   Think of applications! (Keep your answer to this part to < 100 words).
   Ans
   1. Docker: Deploy, manage, and test a small application on a single machine with a consistent development environment, e.g., a web application or games.
   2. Container: Container runtime is preferred when you want to get lower overhead, access to lower-level container internals, or more seamless interaction with other tools, e.g., a basic web server or Redis cache, for improved resource efficiency and better interaction with Kubernetes.
   3. VMs: need to run applications on different platforms, e.g., testing applications in different OS platforms.
   4. K8s: run large-scale distributed systems, e.g., Hadoop, Apache Spark.

   c. Give two key differences between a K8s node, K8s pod, and K8s cluster.
   Ans:
   Difference in functions: K8s node is a physical node or a VM running an OS and can run any program. K8s pod is a container (or a bundle of containers) that is built to run one particular program or task. K8s cluster is a collection of K8s nodes (nodes need not be identical)

Difference in ownership: K8s pod is typically managed by the application developer who write the program. K8s nodes and what they run are typically automatically managed by an orchestrator like Kubernetes. At the highest level K8s cluster is managed by the infrastructure engineer (devops) who decides the number and types of nodes that the cluster consists of.

4. (Solution and Grading by: Hanbo Guo)

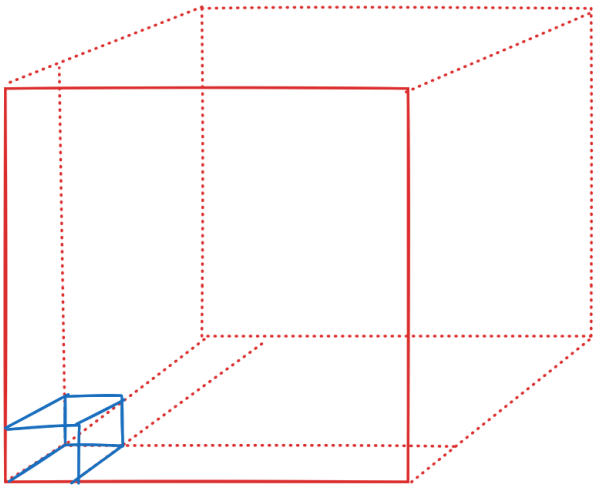
   Map
   - 40 machines X 4 containers = 160 tasks at most can be run at once.
   - So the 240 Map tasks will run in two “waves”: 160 tasks first (taking **15 seconds**), and then the remaining 80 tasks (taking **another 15 seconds**).

   Shuffle
   - The question says shuffle traffic starts only after all Map tasks are completed.
   - The total shuffle traffic = 20MB X 240 = 4.8GB = 38.4 Gb.
     - (this is an optional step) Since only 1/N-th of this traffic will go over the network (N=number of cluster machines=40), total tr=affic is 38.4Gb X 39/40 = 37.44Gb
   - At 2 Gbps, this takes **18.7seconds** to transfer.

   Reduce
   - There are 180 Reduce tasks. Again with 40X4 = 160 containers, we need two “waves” of Reduce tasks: first wave with 160 tasks takes **35 seconds**, and second wave with 20 tasks takes **another 35 seconds**.
   - Reduce time (35 seconds) includes time to write output to HDFS.

   Total time = 15 seconds * 2 + 18.7 seconds + 35 seconds * 2 = 118.7 seconds.
   We will accept answers that are ±1 second away from the above answer.

5. (Solution and Grading by: Talha Waheed)



a) L = M+K+R-3+1 = M+K+R-2

b) If no pingers are alive when the node goes down, completeness is violated. Thus the answer is the same, M+K+R-2

c) This is a trick question, as it's impossible to guarantee accuracy in an asynchronous system with a SWIM/ping based protocol or a heartbeating protocol, because message delays/drops make it impossible to distinguish a failed process from a lossy/slow one--hence since one has to set *some* timeout for the failures to be detected, a process that is slower than this timeout will be falsely detected (no matter what you select the timeout to be). Note that merely saying "Completeness is true, therefore accuracy cannot be guaranteed" without any further explanation, is insufficient and will not result in full points.

6. (Solution and Grading by: Nishant Sheikh)
    a. For Push gossip, once a group is infected, spreading there takes O(log(K)) time. However, for a group to be infected requires the root to infect it. Since the root selects only 1 target per round (1 out of M - imagine a biased coin with heads probability 1/M), a random group takes on average O(M) rounds for at least one of its nodes to be infected (expected number of coin flips of a 1/M biased coin to turn up first heads is = M flips). Thus the latency is O(M)+O(log(K)).
    It is also ok if you calculated the “whp” latency of the group dissemination as O(Mlog(M)) (e.g., via the Coupon Collector’s problem) instead of the average O(M) term.
    b. For Pull gossip, since all nodes know the root, one can imagine each group as consisting of K+1 nodes (with the root included). Pull gossip in this group takes O(log(K)) time.
    If you calculated the “whp” latency, you would get O(log(M)) + O(log(K)). Intuition for the O(log(M)) group dissemination component comes from the detail that the pulls happen in parallel. (For math details, see https://math.stackexchange.com/q/26167)
    c. Pull is faster than Push. This is because the root is the bottleneck in the push, but pull removes the bottleneck.
    d. The pull gossip in this setting overwhelms the root. On average the root receives O(M) pull requests every gossip round, which can be prohibitively large.

7. (Solution and Grading by: Gabriella Xue)
    a. **Pro**: Entries can be cleaned up right after detection, so memory usage is small especially under high churn (note that failure detection time does *not* change by setting tcleanup=0, as failure detection time still remains tfail, independent of tcleanup). **Con**: Ghost entries (referring to failed nodes) may circulate forever in membership lists. (ghost entries are separate from false positives and false negatives)
    b. **Pro**: Faster detection time (no need to wait for suspicion timeout). **Con**: Higher false positives (i.e., no second chances—for any non-faulty process Pi, if any one process Pj mistakenly detects Pi as failed, Pi will be forced to leave the system)
    c. **Pro**: Avoids overhead of shuffling the membership list (when end reached), and of linear traversal of the list. **Con**: Worst case detection times become unbounded.

8. (Solution and Grading by: Tianchen Wang)

// Solution is for 3 candidates, can be generalized to any number of candidates

M1(key=null, V=(V1, V2, V3)): // main idea: output for each pair the voter's preferred ordering
- output <lex_sorted(V1, V2), 1 if order unchanged else -1> // "order unchanged" means lexicographic ordering and voter's preferred ordering are the same.
- output <lex_sorted(V1, V3), 1 if order unchanged else -1>
- output <lex_sorted(V2, V3), 1 if order unchanged else -1>

R1(key=(V1, V2), value_array): // main idea: given a voter's preferred ordering, output the "winner" of that head to head battle. Note that
sum_ints = sum of elements in value_array
if sum_ints > 0
    - output <V1, 1>
if sum_ints < 0
    - output <V2, 1>

M2(key=V, value=1): // swap key and value, so that we can pipe everything to one reduce task next
- output <1, V>

R2(key=1, values={V1, V2, …}): // single reduce task to find the winner.
voter_map = map()
for all V in value, voter_map[V] += 1
keys = argmax(voter_map) // this will output the key with max value, list of keys if there are multiple max values
if len(keys) == 1:
    - output <keys[0], "Condorcet winner!">
else:
    - output <votermap[keys[0]], "Highest Condorcet Counts">

9. (Solution and Grading by: Kartik Ramesh)

The first Map Reduce processes the dataset to find the following and followers of each user, and applies conditions 1-3.

```
# input (a, b)
map1(key, value) {
        output(a, (b, "OUT")) // Outgoing link from a
        output(b, (a, "IN"))  // Incoming link to b
}

# key := user, values := [ ]((string, string))
reduce1(key = user, values) {
        following := create list from values with value "OUT".
        followers := create list from values with value "IN".

        areFollowingFollowers = true if all users in following are in followers else false

        if (len(followers) >= 10m && len(following) < 10 && areFollowingFollowers) {
                output(key, ("CANDIDATE", following)) // potential answer.
        }

        output(key, ("FOLLOWER", followers)) // To identify the special user U.
}
```

The second Map Reduce applies condition 4. on the filtered candidates.

```
func map2(key=user, value) {
        output(1, (key, value))
}

# values = [ ](user, (string, [ ]string))
func reduce2(key = 1, values) {
        candidates_and_following := filter users and following from values
                                        with value “CANDIDATE”
        users_and_followers:= filter users and followers from values with value “FOLLOWER”

        user_with_max_follows = find user with maximum followers in users_and_followers

        for candidate in candidates:
                if !user_with_max_follows in candidates.following:
                        output(candidate.user_id)
}
```

10. (Solution and Grading by: Madhav Jivrajani)

For D1

// Map1 filters post entries for Monday or Friday.
Map1(a, p, wd:hh:mm:ss) -> emit(a, p) if wd == “Mon” OR wd == “Fri”
Reduce1: output identity // file format: a,p

Output referenced as MR1

For D2

Map2(a, p, wd:hh:mm:ss) -> emit(p, 1)
Reduce2(key=p, values=[1, 1…]) -> emit(p, like_count)

Output referenced as MR2

For D3

Map3(a) -> emit(a, “TENNIS”)
Reduce3: output identity // file format: a,TENNIS

Output referenced as MR3

Join MR1 and MR3

Map4: output identity
Reduce4(key=a, values=[p1, p2, …, maybe TENNIS]) {
    if not “TENNIS” in values:
        return
    for p in values:
        if p != “TENNIS”: emit(p, a)
}

Output referenced as MR4

Join MR2 and MR4

Map5: output identity
Reduce5(key=p, value=[(a, like_count)]) -> emit(a, like_count)

Calculate total likes per user

Map6: output identity
Reduce6(key=a, values=[like_counts)] -> emit(a, total_like_count)

Calculate top k

```
Map7(key=a, value=like_count) -> emit(1, (a,like_count))
// Reduce7 has no parallelism; single reducer.
Reduce7(key=1, values=[(a,total_like_count)s]) {
        topK = get_topk_user_ids_for_counts(values) // list of user IDs.
        emit(topK) // write as \n separated values to file.
}
```

Note: we might not need MR3. We can write a reader impl. that reads MR1 and D3 and uses NULL values instead of the TENNIS sentinel.