

Midterm C Solutions: CS425 FA 26

Note to Students: Recommended solutions for Midterm C (FA26) are below. For many questions, alternate solutions are possible and reasonable correct solutions will be accepted during grading.

1. (Solution by: Will Verified by: Parham)

a. Q1: C

- i. $PUE = \text{total facility energy} / \text{IT equipment energy}$. IT equipment used $15000 - 3000 = 12000$ kWh, so $PUE = 15000 / 12000 = 1.25$

b. Q2: E

- i. BitTorrent uses local rarest first, meaning the client fetches the block with the fewest copies among its neighbors. Across A, B, C, D, block 0 has 2 copies, block 1 has 3, block 2 has 2 and block 3 has 2 and block 4 only has 1 (at C). Block 4 is the rarest so X fetches it first.

c. Q3: C

- i. A higher Lamport timestamp does not imply causality. Two concurrent events on different processes can have different Lamport timestamps, yet their vector timestamps are incomparable, so e2's vector timestamp is not guaranteed to be higher.

d. Q4: D

- i. Sorting occurs on both sides: output data from each Map worker are sorted, and input data to each Reduce worker are merge-sorted before they are processed.

e. Q5: D

- i. A read only overlaps a write when $R + W > N$. C100 reads with ONE, so its read may hit a replica that isn't in the write quorum and return a stale value. That breaks choice A since C100 may not see its own write, and it breaks choice B because C100 may not see anyone's write

f. Q6: C

- i. Neither vector is less than or equal to the other in every entry. e1 is smaller in the first two positions and larger in the last two, so the timestamps are incomparable, which means the events are concurrent.

g. Q7: A

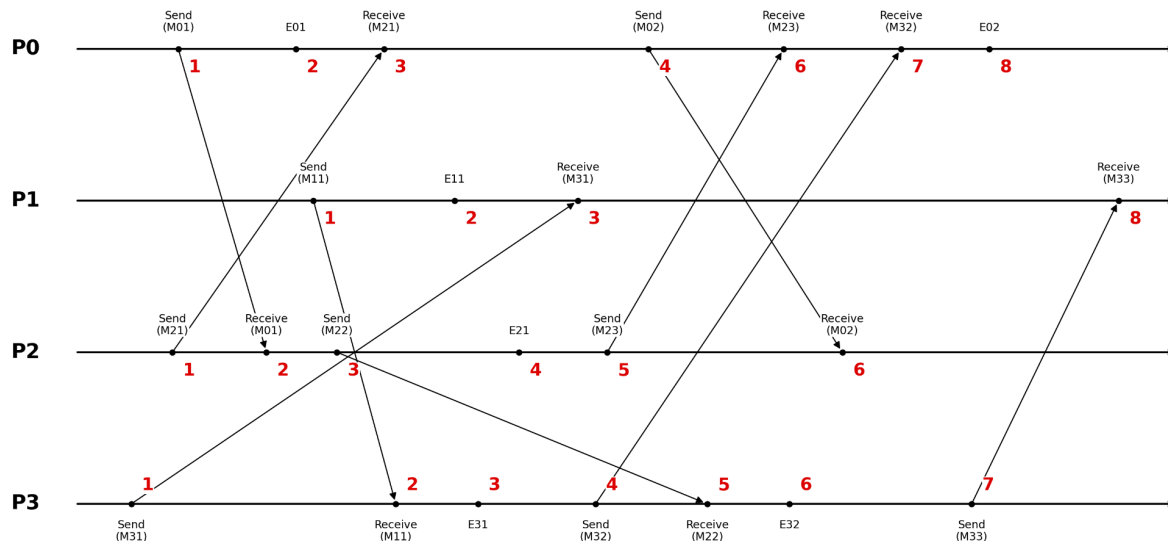
- i. MTTF scales inversely with number of servers. Going from 10k to 30k servers triples the failure rate so MTTP drops from $6\text{min} / 3 = 2\text{min}$

- h. Q8: C
 - i. Chord routes each lookup deterministically using finger tables, reaching the node in $O(\log N)$ hops.
- i. Q9: B
 - i. In CAP, consistency means every read sees the latest acknowledged write, as if there were a single up-to-date copy of the data.
- j. Q10: C
 - i. The client gradually slows its clock until the server's time catches up. This avoids moving time backward, which could disrupt event ordering and applications.

2. (Solution by: Wei Verified by: Will)

a.

(a) Lamport timestamps



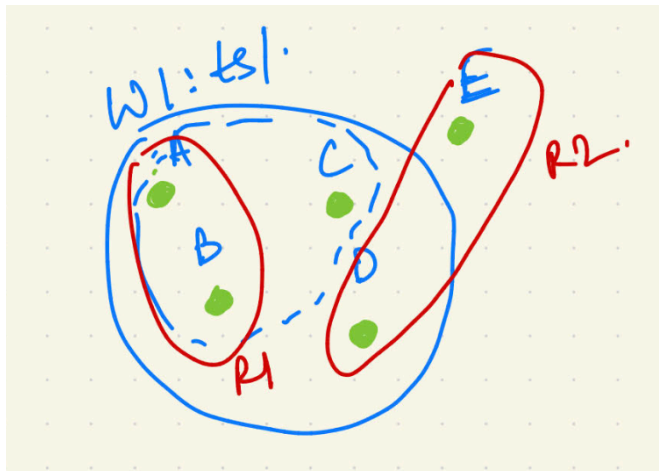
- b. **First, read the instructions & hint carefully, and recall the vector timestamp update rule!** The third element is 5. It corresponds to P2, which increments its own vector component for every local, send, or receive event. Through Send(M23), P2 has five events: Send(M21) → Receive(M01) → Send(M22) → E21 → Send(M23). The full vector timestamp at Send(M23), in the order (P0, P1, P2, P3), is (1, 0, 5, 0). P2 knows about P0's first event through M01 and has received no information from P1 or P3.

3. A. The standard Chord with exponential steps covers the ring space in $O(\log N)$ hops because each step halves the remaining distance. The worst case routing complexity for our Fibonacci variant is $O(N)$. Theoretically we could build a Fibonacci chord with each interval bigger than the largest offset, just like the chord in the problem.

B. Cassandra

(i) **Yes.** The last write that completed before the read began, with timestamp t_w , was acknowledged by W replicas. A read gets responses from R replicas. Since $R + W > N$, those two sets must overlap in at least one replica. The read returns the newest timestamp among its R responses, so it returns a version with timestamp at least t_w . Because a single coordinator assigns all the timestamps, they follow the order the writes were issued.

(ii) **No.** Since multiple reads and writes can be in progress for a key K at the same time, the first read can see a write that is in-progress and has reached fewer than W replicas. The second read might reach a disjoint subset from the first read and be overlapping with replicas that are in the write quorum but haven't yet received the write.



Consider the following example. $N = 5$, $W = 4$, $R = 2$.

Replicas A to E all start with (k_0, v_0) at timestamp 0. Consider the following steps.

1. Write $W1$ (k_0, v_1) at timestamp ts_1 starts and reaches A, B, C: 3 acks, still short of $W = 4$
2. Read $R1$ asks A and B, gets $\{v_1, v_1\}$, returns **v_1** , and completes
3. Read $R2$ starts, asks D and E, gets $\{v_0, v_0\}$, returns **v_0**
4. $W1$ reaches D and completes

Read $R2$ began after $R1$ completed and returned an older version. The figure above should clarify better.

4. (Solution by: Xinying Verified by: Tianyi)

A: We use two MapReduce jobs.

MapReduce Job 1: Identify failures involved in suspicious bursts

Input: Security logs.

Map:

For each log record (timestamp, user-id, ip-address, status)

Emit: (user-id, (timestamp, ip-address, status))

Reduce:

1. For each user-id, receive all of its records and process them in increasing timestamp order.
2. Scan the records while maintaining the current consecutive sequence of FAILURE attempts.
 - a. A SUCCESS resets the current FAILURE sequence.
 - b. For each FAILURE, add it to the current sequence.
 - c. Whenever the current sequence contains at least 4 consecutive FAILURE attempts, check the most recent 4 failures.
 - d. If "**timestamp of the latest**" - "**timestamp of the fourth latest**" ≤ 120 , then those 4 attempts form a suspicious burst.
 - e. Mark each of those failure records as belonging to a suspicious burst, if not already.
 - f. For every failure record marked as belonging to a suspicious burst, emit:((ip-address, user-id), 1)

Output: ((ip-address, user-id), 1), the emitted 1s represent individual failed login attempts that belong to at least one suspicious burst.

MapReduce Job 2: Count suspicious-burst failures and identify high-threat IPs

Input: Output of MapReduce Job 1

Map:

For each ((ip-address, user-id), 1)

Emit (ip-address, (user-id, 1))

Reduce:

For each ip-address, collect all (user-id, 1) records:

1. Group the records by user-id and sum the values to obtain the total number of failed login attempts belonging to suspicious bursts for each user: (user-id, attempt_count)
2. If the IP has fewer than 3 distinct user-ids, emit nothing.
3. Otherwise, construct a comma-separated list: user-id:attempt_count for all targeted users.
4. Emit: (ip-address, "BOTNET_TARGETS:" + comma-separated list)

Output: The total number of FAILURE attempts from each IP against each user over the entire log period.

Here's a more pseudo-code like version of this:

1st stage

```
record = {timestamp, user-id, ip-address, status}
```

```
map(key offset, value record) {  
    emit(record.user-id, {record.timestamp, record.ip-address, record.status});  
    // time-ordered sequence per user-id  
}
```

```
reduce(key user_id, value records) {  
    burst_ip_cnts = map<ip-address, integer>;  
    current_failures = [];           // current contiguous run  
  
    // Helper function to evaluate a contiguous run of failures  
    function process_failure_block(failures) {  
        if (failures.len() < 4)  
            return;  
  
        // Track whether each failure belongs to a valid burst window  
        in_burst = array of boolean (length = failures.len(), default false);  
  
        // 2-pointer sliding window over the failure sequence  
        left = 0;  
        for right from 0 to failures.len() - 1 {  
            // Shrink window from the left if time span > 120 seconds  
            while (failures[right].timestamp - failures[left].timestamp > 120) {  
                left++;  
            }  
  
            // If the current valid 2-minute window has >= 4 failures, mark all of them  
            if ((right - left + 1) >= 4) {  
                for i from left to right {  
                    in_burst[i] = true;  
                }  
            }  
        }  
  
        // Count valid burst failures aggregated by IP address  
        for i from 0 to failures.len() - 1 {  
            if (in_burst[i]) {  
                burst_ip_cnts[failures[i].ip-address]++;  
            }  
        }  
    }  
  
    for record in records {  
        if record.status == "SUCCESS" {  
            process_failure_block(current_failures);  
            current_failures.clear();  
        }  
    }  
}
```

```

    } else {
        current_failures.append(record);
    }
}

// Process remaining failures at end of the loop
process_failure_block(current_failures);

// Emit accumulated burst failure counts per IP
for (ip, count in burst_ip_cnts) {
    emit(ip, {user_id, count});
}
}

```

2nd stage

```

record = {ip, user_id, count}
map(key ip-address, value record) {
    emit(ip-address, record);
}

reduce(key ip-address, value records) {
    userid_totals = map<user_id, integer>;

    for record in records {
        userid_totals[record.user_id] += record.count;
    }

    // IPs that targeted at least 3 distinct user-ids
    if (userid_totals.keys().size >= 3) {
        targets_list = [];
        for (user_id, count in userid_totals) {
            targets_list.append(user_id + ":" + count);
        }

        emit(ip-address, "BOTNET_TARGETS: " + join(targets_list, ", "));
    }
}

```

B:

Advantage: Reducer output can remain on the reducer's local disk, avoiding the need to write the intermediate output to a distributed file system and potentially reducing network I/O and intermediate-data transfer.

Disadvantage: Intermediate output is less fault-tolerant. If the machine containing a reducer's local output fails before the next MapReduce job reads it, that output may be

lost. It also makes the next job dependent on the availability of the machines that produced the previous output.

5. (Solution by: Parham Verified by: Xinying)

- a. P_j 's monitor set is k predecessors plus a random extra that may well be zero. So the smallest possible monitor set has size k (pt = 2), and the cheapest way to hide a failure is to kill P_j together with its k predecessors (pt = 2). Completeness holds for up to k simultaneous failures and can break at $k+1$. So the answer is $M = k+1$ (pt = 3).
- b. No (pt = 3). The system is asynchronous, so there is no bound on how long a heartbeat may take to arrive or on how long a healthy process may be stalled. A correct process that pauses for some reason, gets descheduled under load, or sits behind a briefly congested link will miss its deadline at every one of its receivers at once and be declared dead while running perfectly well. Therefore, false positives are possible, and the algorithm is not 100% accurate (pt = 4).
- c.
 - i. Best case: nobody randomly picked $P_j \Rightarrow k$ heartbeats per period = $k / 2$ per second. (pt = 1 for answer + 1 for explanation)
 - ii. Worst case: every process that is allowed to pick P_j did. All $N-(k+1)$ eligible processes choose it, giving $k + (N-k-1) = N-1$ per period = $(N-1) / 2$ per second. (pt = 1 for answer + 1 for explanation)
 - iii. $x(i,j) = 1$ if p_j in $\text{recv}(p_i)$ and $L_j = \text{Sum}(x(i,j))$ for all $i \neq j$. So $E[L_j] = \text{Sum}(E[x(i,j)])$ for all $i \neq j = \text{Sum}(\text{Pr}[p_j \text{ in } \text{recv}(p_i)])$ for all $i \neq j$. For the k predecessors $\text{Pr}[p_j \text{ in } \text{recv}(p_i)] = 1$ and for everyone else $\text{Pr}[p_j \text{ in } \text{rand}(p_i)] = C(n-1, k-1) / C(n, k) = k / n = k / (N - k - 1)$. So we have $E[L_j] = k + (N-k-1) * (k / (N-k-1)) = k+k = 2k$ per period, so the answer is k per second. (pt = 1 for answer + 1 for explanation)