CSE 461 Homework 3

Ken Lin

Total points: 48 points assumed

1, (20 points)

A DHT Chord network uses 4 bits (i.e. m = 4) to identify machines and keys of entities. At a certain time, machines with identifiers 2, 5, 9, and 11 are attached to and active in the network.

A, Draw a diagram to show the machine ids and keys of the network.

B, Find the finger table of each of the machines.

A & B scanned below.

Finger Tables transcribed:

FT2 =

1	5
2	5
3	9
4	11

FT5 =

1	9
2	9
3	9
4	2

FT9 =

1	11
2	11
3	2
4	2

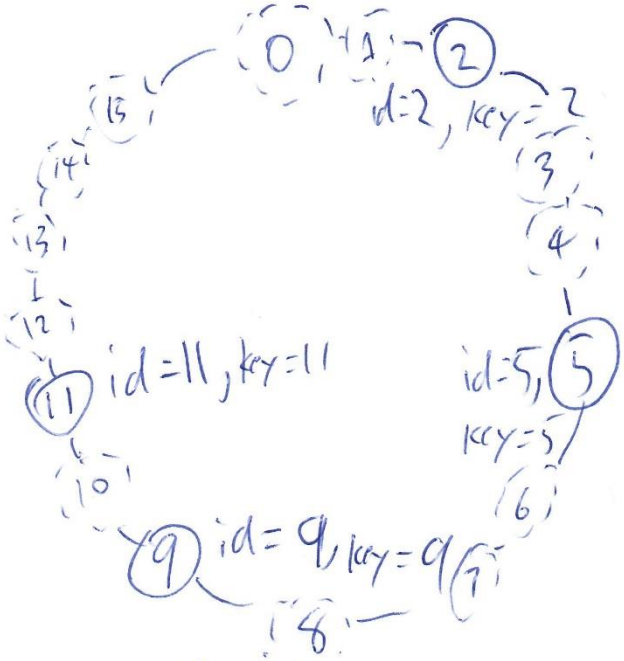
FT11 =

1	2
2	2
3	2
4	5

$2^4 = 16$ 0–15 HW Q 1

a)

$m = 4$



b)

FT_2 ($2 + 2^{i-1}$)

1	5
2	5
3	9
4	11

FT_5 ($5 + 2^{i-1}$)

1	9
2	9
3	9
4	2

FT_9 ($9 + 2^{i-1}$)

1	11
2	11
3	2
4	2

FT_{11} ($11 + 2^{i-1}$)

1	2
2	2
3	2
4	5

C, An application running in node 11 is looking for the entity with key value 7. Find the route the system takes to get to the node that has the entity. Show your steps clearly and draw the route on your diagram.

Scanned solution below:

Note that steps 2+ in scanned solution is incorrect, please see step 2+ in transcribed steps for corrected solutions.

Transcribed steps:

Step 1, look up a index j which fulfills FT11[j] <= 7 < FT11[j+1]. Unfortunately, no such index j exists. (7 > 5, the value inside the table at the largest index, FT11[4].) So we go to FT11[4], which is node 5. We forward the request to node

Step 2, Look in FT5 for FT5[j] <= 7 < FT5[j+1]. However, no node fulfills this condition. So, because FT5[4] <= 7, we forward the request to node FT2[4], which is 2. We make a branch: (a).

However, node 5 is less than 7 which is also less than FT5[1], so we also forward the request to FT5[1], which is 9. We make a branch: (b).

Step 3:

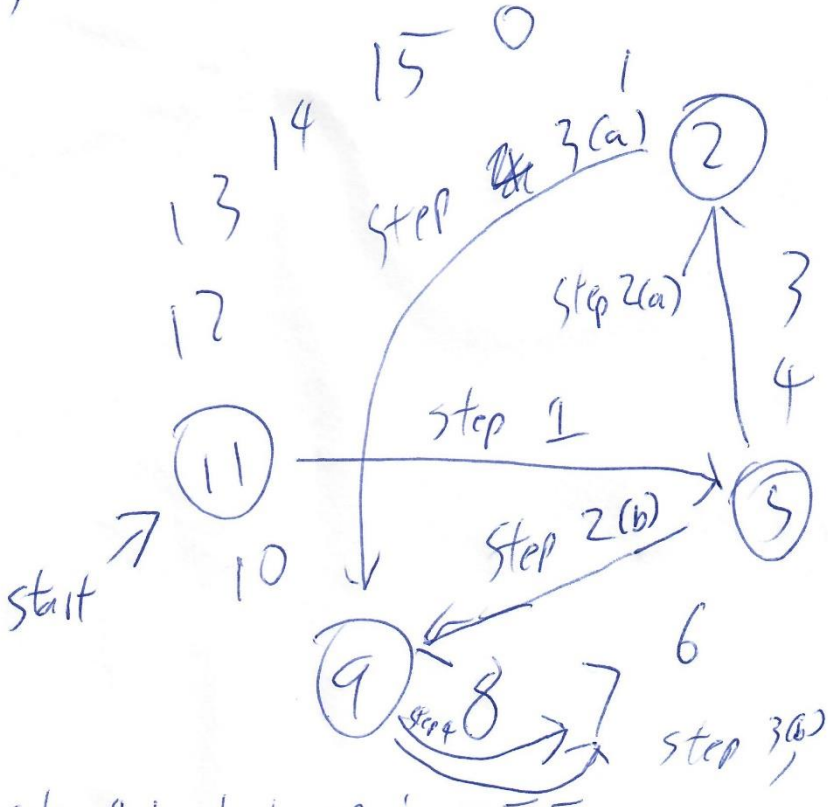
(a), Look up in FT2 for FT2[j] <= 7 < FT2[j+1], j = 2 fulfills this condition. 5 <= 7 and 7 < 9. We forward branch (a)’s request to node 9.

(b), Node 9 is the successor of 7, so we find key 7 at node 9.

Step 4:

(a) Node 9 is the successor of 7, so we find key 7 at node 9.

c) looking for key = 7



step 1: look up in FT_{11} a node where $FT_{11}[j] \leq 7 < FT_{11}[j+1]$. But $7 > 5$, the last node in FT_{11}, so we go to node 5. $j = 4$

Step 2: Look in FT_5 for $FT_5[j] \leq 7 < FT_5[j+1]$. Node 9 fulfils this condition. $j = 3$

Step 3: Node 9 is succ(7), so we found 7.

2, (10 points)

Explain what a content delivery network (CDN) is? How does it work? What advantages it could provide to users as compared to a conventional network?

CDN is a service that accelerates internet content delivery. That is, it makes it faster to fetch data from a server.

Normally, in order to load the data from a server a request has to be sent to the server and then the server sends a reply. If the server is far from the user then the round trip time is significant. A CDN, however, reduces the distance between the data from a server (the content), by distributing the content in many locations around the world. When a user tries to access the content a request is sent to the server, the server distributes the content to the locations around the world, and the user can now fetch the contents from a closer location than the main server.

The main advantage of a CDN is that it makes the user's website faster. That is, a website using CDN will load faster compared to 'normally.' Additionally, because content is fetched from many locations rather than always from the main server, the main server has less load. Because the server has less load, it has greater uptime available to respond to users. Also minor security benefits due to this.

3, (10 points) Consider the behavior of two machines in a distributed system. Both have clocks that are supposed to tick 1000 times per millisecond. One of them actually does, but the other ticks only 990 times per millisecond. If UTC updates come in once a minute, what is the maximum clock skew that will occur?

Assuming that an UTC update sets each machine to 0 ticks rather than simply setting both machines to the same second/millisecond. That is at the start of a minute both machines are at tick 0. (Otherwise, for example, it might be possible that the faster machine was at tick 999 while the slower machine was at tick 0 at the start of a minute, and 1/1000 of a millisecond later the faster machine would be at tick 1000, almost a full millisecond ahead of the slower machine. Despite both machines being at the same millisecond when the minute started.)

In this case, both systems simply tick for 60 seconds before the next UTC update, and both machines start at tick 0.

The faster machine ticks 1000 times/ms * 60s/minute * 1000ms/s = 60,000,000 times.

The slower machine ticks 990 times/ms * 60s/minute * 1000ms/s = 59,400,000 times.

The maximum clock skew is then 60,000,000 – 59,400,000 = 600,000 ticks.

4, (8 points)

If each process uses a different value for d in the Lamport's clock and vector clock equations, will the logical clocks and vector clocks schemes satisfy the total order relation => and the relation:

a → b iff ta < tb

Explain your argument in detail.

(Lamport's clock by itself would fail to satisfy the relation a → b iff ta < tb.)

With the vector clock scheme, the total order relation would be satisfied. Additionally, a would happen before b iff ta < tb. (As long as d > 0, even though they're different for all processes.) The reason is because Lamport's clock deals with relative ordering, so it doesn't matter whether the clock is advanced by 1, 2, 3.14, or any arbitrary positive value so long as implementation rules of vector clocks are followed.

Because Lamport's clock & vector clock is a logical clock, the it's the relative order of events that matters rather than the absolute value of the entries in the vector clock timestamps. Changing d between processes would not affect any of the requirements of the total order relation or the relation a → b iff ta < tb. The vector clock implementation is enough for the total order relation and the relation a → b iff ta < tb, and changing d between processes do not change it, therefore each processs using a different value for d would also satisfy the total order relation and the relation a → b iff ta < tb.

Conditions of total order relation:

1. for any two events **a** and **b** in a process P_i,
   if a occurs before b, then

   $$C_i(a) < C_i(b)$$

This is satisfied regardless of the value of d so long as the implementation rule:

two successive events in P_i

$C_i = C_i + d$ ($d > 0$)

if a and b are two successive events in P_i and a → b then

$C_i(b) = C_i(a) + d$ ($d > 0$)

Is followed. Events that occur after an event would have a greater value for Ci regardless of whether d is 1 or 100 so long as the rule is followed, so this condition isn't an issue.

2. if **a** is the event of sending a message **m** in P_i
   and **b** is the event of receiving the same message **m**
   at process P_j, then
   $C_i(a) < C_j(b)$

Similarly, as long as the implementation rule is followed:

event a: sending of message m by process Pi,

timestamp of message m : tm = Ci(a) then

Cj = max (Cj, tm + d) d > 0

Because Cj, the timestamp of the message receipt by the receiving process is the maximum of Cj, the time at the receiving process or the time of the message + d, it wouldn't matter if two processes used different values for 'd.'

In the case that the sender uses a d that's so large they will always be 'ahead' of the receiving process, then Cj will always be tm + d, which means Cj will always have occurred after the message being sent. (Because tm + d is always larger than tm for d > 0, and because we're using an very large d tm + d will always be larger than Cj, meaning everything that happened before the message was sent will have a timestamp smaller than tm + d, which satisfy the relation.)

In the case the sender uses a d that's so small they will always be 'behind' the receiving process, then Cj will always be Cj. Because Cj uses a significantly larger d, Cj will always have a massively larger timestamp, so all the messages it receives will have came after the messages are sent. This satisfies the relation.

In the case where the difference in d is small enough that Cj can be greater than or less than tm + d, in the cases where tm + d > Cj then Cj would be updated to tm + d, fulfilling the requirement, and in the cases where Cj > tm + d, Cj would stay Cj, again fulfilling the requirement.

Different values for 'd' do not affect this condition.

In addition, total order relation requires:

a is any event in process P_i
b is any event in process P_j **a** => **b** iff
either $C_i(a) < C_j(b)$
or $C_i(a) = C_j(b)$ and $P_i \prec P_j$ (e.g. $P_i \prec P_j$ if $i \leq j$, to break ties)

This is an implementation requirement to break ties, so it's not affected by the choice of 'd.' While it's true that a process will an exceeding large 'd' will seem to always be later than a process with an exceedingly small 'd,' the fact that the large 'd' is ahead in timestamp doesn't matter unless a message is passed, in which case the earlier implementation would force the correct logical ordering. (Requirement 2.)

All processes using different 'd' therefore does not affect the total ordering relation, and Lamport's logical clock is enough to fulfill the total ordering relation, therefore all processes using different 'd's fulfills the total ordering relation.

Now for $a \rightarrow b$ iff $ta < tb$:

As mentioned before, Lamport's clock by itself isn't enough for $a \rightarrow b$ iff $ta < tb$, we need the vector clock. Further, the vector clock implementation isn't affected by choosing different 'd' for all processes. Therefore, because the vector clock implementation is enough to satisfy $a \rightarrow b$ iff $ta < tb$, the vector clock implementation with different 'd' for all processes satisfies $a \rightarrow b$ iff $ta < tb$.

Implementation rules:

1. two successive events a, b in process P_i:

$$C_i(b)[i] = C_i(a)[i] + d \quad (d > 0)$$

Similar to rule 1 for the total order relation, this rule is satisfied regardless of the choice of d. Within the same process the choice of d doesn't matter as long as $d > 0$, events that occur later will have larger timestamps than events that occurred earlier.

2. event a at P_i sending message m to process P_j with receiving event b; vector timestamp $\mathbf{t_m = C_i(a)}$ is assigned to m; on receiving m, P_j updates C_j as follows:

$$\text{all } k,\ C_j(b)[k] = \max(C_j(b)[k], t_m[k])$$

'd' isn't even in this rule, so obviously each processes having different 'd' would not affect the implementation of this rule. The reason the receiving timestamp is merely the max of Cj(b)[k] or tm[k] rather than tm[k] + d is because the local clock,

Cj(b)[k], is always at least ‘d’ greater than Ci(a)[k] because the event of receiving a message must have happened after the event of process j sending a message to process ‘i’ which updated process i’s vector clock table. (Or else process j never sent a message to process ‘i’ at all, in which case it would *still* be at least d behind, because in that case Ci(a)[k] would be 0 while Cj(b)[k] would be at least d, because the first event in process j has a time stamp of at least d.) Note that even though d is technically hidden in Cj(b)[k] in this rule, the choice of ‘d’ would not affect the relative ordering of events.

The choice of d therefore does not affect the relation a → b iff ta < tb, therefore a vector clock implementation with different ‘d’ for all processes fulfils the requirement a → b iff ta < tb.

The total order relation => and a → b iff ta < tb are therefore both satisfied by a vector clock and Lamport’s clock implementation with different ‘d’ for all processes. Because a logical ordering of events only cares about relative ordering, each clock advancing at a different rate would not affect where each event is placed relative to other events.

5, (10 points)

Suppose Process

P1 has events

e11, e12, e13, e14, e15 e16 e17

P2 has events

e21, e22, e23, e24, e25, e26,

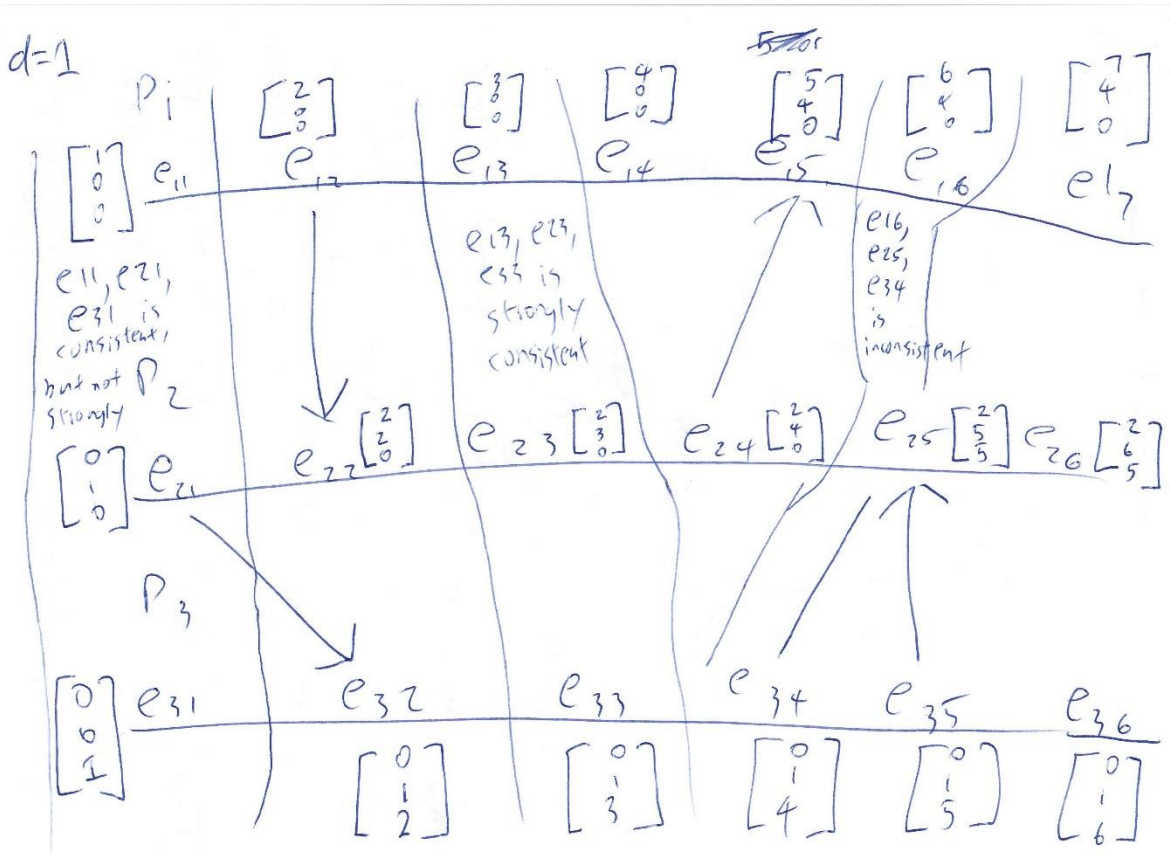
P3 has events

e31, e32, e33, e34, e35 e36

There are message transits from e12 to e22, e24 to e15, e21 to e32, e35 to e25. Suppose the vector time clocks for e11, e21, and e31 are

[1, 0, 0], [0, 1, 0], [0, 0, 1] respectively.

a) Draw a diagram to show all the transitions and events. (An arrow is a message event.)



b) Find the vector clocks of all the events.

Vector clocks transcribed.

P1 =

[1, 0 ,0], [2, 0, 0], [3, 0, 0], [4, 0, 0], [5, 4, 0], [6, 4, 0], [7, 4, 0]

P2 =

[0, 1, 0], [2, 2, 0], [2, 3, 0], [2, 4, 0], [2, 5, 5], [2, 6, 5]

P3 =

[0, 0, 1], [0, 1, 2], [0, 1, 3], [0, 1, 4], [0, 1, 5], [0, 1, 6]

c) Give an example for each of the following:

i) a strongly consistent state

{e13, e23, e33} is strongly consistent.

ii) a consistent but not strongly consistent state

{e11, e21, e31} is consistent, but not strongly. (Message from e21 in transit.)

iii) an inconsistent state

{e16, e25, e34} is inconsistent as a state. A message was received at e25 but P3 did not sent it yet.