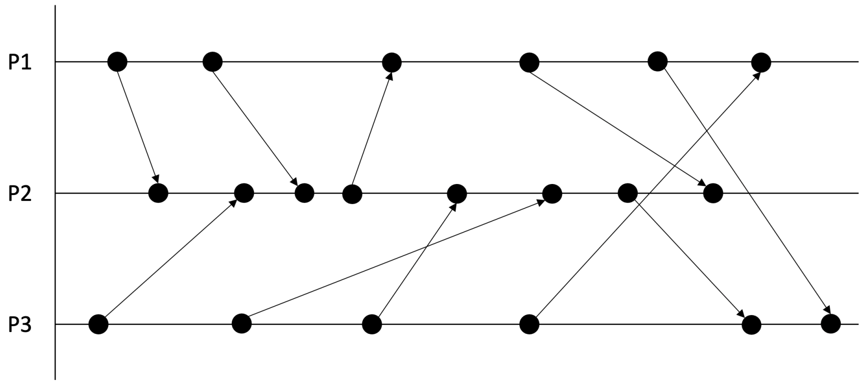
ECLT5820 Distributed and Mobile Systems

Assignment 2 (Topics 5-7)

Due – 11:59pm, 25th Oct. 2021 (Monday)

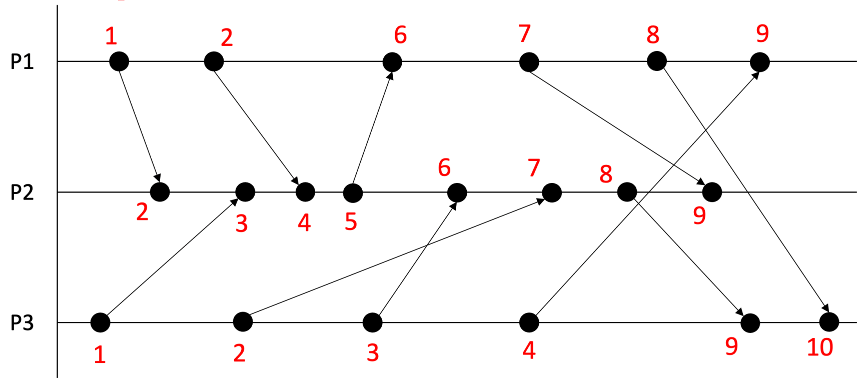
Please send a pdf file to eclt5820@cse.cuhk.edu.hk with Email title and file name "ECLT5820 Asg#2, Your name, Your student ID".

Q1 (**20 points**) Please draw the **Lamport's Timestamp** and **Vector Clock** for the following operations (assuming P1, P2 and P3 all start at 0).

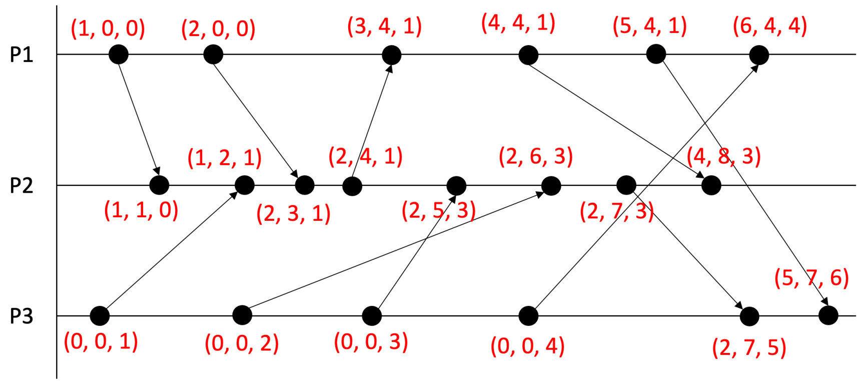


Answer:

Lamport's Timestamp



Vector Clock



Q2 (20 marks) Let us consider the common ordering requirements in distributed systems.

(1) Whether the following statements are true or false? If the statement is false, please justify your answer.

a) Causal ordering includes FIFO ordering.
b) FIFO ordering includes Causal ordering.
c) Total ordering includes FIFO ordering.
d) FIFO ordering includes Total ordering.
e) Total ordering includes Causal ordering.
f) Causal ordering includes total ordering.

(2) In a multi-user fighting game, the players move their figures around a common scene. The state of the game is replicated at the players' workstations and at a server which contains services controlling the game overall, such as collision detection. Updates are multicast to all replicas in the players' workstations. The figures may throw projectiles at one another and a hit debilitates the unfortunate recipient for a limited time. The game incorporates magic devices which may be picked up by a player to assist them.

a) What type of update ordering is required for the event of the collision between the projectile and the figure, and the event of the player being debilitated (which, we may assume, is represented graphically)?
b) What type of update ordering is required for changes in the velocity of the figure and the projectile chasing it?
c) Assume that the workstation at which the projectile was launched regularly announces the projectile's coordinates, and that the workstation of the player corresponding to the figure regularly announces the figure's coordinates and announces the figure's debilitation. What type of update ordering should these announcements be processed?
d) When two players move to pick up a piece at more-or-less the same time, only one should succeed. What type of ordering should be applied to the pick-up-device operation then?

Answer:

(1)

a) True.
b) False. FIFO ordering does not handle the order of messages sent by different processes.
c) False. Total ordering does not guarantee that the delivery order of messages is same to the multicast order.
d) False. FIFO ordering does not handle the order of messages sent by different processes.
e) False. Total ordering does not consider the causal relationship between messages.
f) False. Causal ordering does not consider the order of messages without causal relationships.

(2)

a) Causal
b) Causal

c) Causal
d) Total

Q3 (**15 marks**) A conflict exists when two transactions access the same item, and at least one of the accesses is a write. Let transaction T_1 deposit \$100 to account Alice (A) and \$200 to account Bob (B); transaction T_2 transfer 20% of the balance from Alice to Bob. The followings are three interleaving schedules:

Schedule 1	
T_1	T_2
Read(A)	
A:=A+100	
Write(A)	
	Read(A)
	t:=A*0.2
	A:=A-t
	Write(A)
	Read(B)
	B:=B+t
	Write(B)
Read(B)	
B:=B+200	
Write(B)	

Schedule 2	
T_1	T_2
Read(A)	
A:=A+100	
Write(A)	
Read(B)	
B:=B+200	
Write(B)	
	Read(A)
	t:=A*0.2
	A:=A-t
	Write(A)
	Read(B)
	B:=B+t
	Write(B)

Schedule 3	
T_1	T_2
Read(B)	
B:=B+200	
Write(B)	
	Read(A)
	t:=A*0.2
	A:=A-t
	Write(A)
Read(A)	
A:=A+100	
Write(A)	
	Read(B)
	B:=B+t
	Write(B)

(1) Are these schedules serially equivalent interleavings? If so, what are they serially equivalent to? Please explain.

(2) Assuming transactions T_1 and T_2 will commit only after the last operation (i.e., Write(A) or Write(B) for T_1 and Write(B) for T_2). Which transaction(s) are strict transaction(s)? Identify which of the following problems is encountered in any of the three schedules (please indicate all such problems in these transactions, if any):

a) dirty reads problem
b) premature writes problem (assuming 'overlaps' are not allowed for write operations of different transactions)

Please explain your answers in detail.

Answer

(1) Both Schedule 1 and Schedule 3 are not serially equivalent interleavings. Schedule 2 is serially equivalent to T_1 happens before T_2.

(2) Schedule 2 is a strict transaction

a) Dirty read problem occurs to Schedule 1 twice as follows:
(i) T_1 writes A and aborts, then T_2 will read dirty A.
(ii) T_2 writes B and aborts, then T_1 will read dirty B.

Dirty read problem occurs to Schedule 3 once:
(i) T_2 writes A and aborts, then T_1 will ready dirty A.

b) Premature write problem occurs to Schedule 1 twice:
(i) T_1 writes A, T_2 writes A, then T_1 aborts, before-image of A overwrites A.
(ii) T_1 writes B, T_2 writes B, then T_1 aborts, before-image of B overwrites B.

Premature write problem occurs to Schedule 3 twice:
(i) T_1 writes B, T_2 writes B, then T_1 aborts, before-image of B overwrites B.
(ii) T_2 writes A, T_1 writes A, then T_2 aborts, before-image of A overwrites A.

Q4 (15 marks) An enhanced two-phase commit protocol has the following parts:

Step 1: The same as the first phase of the two-phase commit protocol.

Step 2: The coordinator collects the votes and makes decisions; if the decision is *No*, it aborts and informs participants that voted *Yes*; if it is *Yes*, it sends a *preCommit* request to all the participants. Participants that voted *Yes* wait for a *preCommit* or *doAbort* request. They acknowledge *preCommit* requests, and carry out *doAbort* requests.

Step 3: The coordinator collects the acknowledgments. When all are received, it *Commits* and sends *doCommit* to the participants. Participants wait for a *doCommit* request. When it arrives, they *Commit*.

Explain how this protocol avoids delay to participants during their 'uncertain' period due to the failure of the coordinator or other participants comparing with the two-phase commit protocol. Assume that communication does not fail.

Answer:
This enhanced two-phase commit protocol is called three-phase commit protocol. In the two-phase commit protocol, the "uncertain" period occurs because a worker has voted *Yes* but has not yet been told the outcome (It can no longer abort unilaterally).
In the three-phase commit protocol, a worker's "uncertain" period lasts from when the worker votes *Yes* until it receives the *PreCommit* request. At this stage, no other participants can have committed. Therefore, if a group of workers discover that they are all "uncertain" and the coordinator cannot be contacted, they can decide unilaterally to abort.

Q5 (15 marks) Please answer the following two questions.

1) Apply two-phase locking protocol to the schedules below by adding statements like R-lock(A), W-lock(A), and unlock(A).

T_1	T_2
read(A)	
read(B)	
	read(B)
read(C)	
	read(A)
	read(C)
	write(D)
	write(A)
write(B)	
write(A)	

2) In this 2-phase locking schedule, are there any problems? If yes, explain problem in detail, and propose two ways to solve them (1) by changing the interleaving schedule (2) by changing the lock mode (but without directly changing the inter-leaving schedule).

Answer:
1)

T_1	T_2
R-lock(A)	
read(A)	
R-lock(B)	
read(B)	
	R-lock(B)
	read(B)
R-lock(C)	
read(C)	
	R-lock(A)
	read(A)
	R-lock(C)
	read(C)
	W-lock(D)
	write(D)
	W-lock(A)
	write(A)
	unlock(A)
	unlock(B)
	unlock(C)
	unlock(D)
W-lock(B)	
write(B)	
W-lock(A)	
write(A)	
unlock(A)	
unlock(B)	
unlock(C)	

2)
(1) Yes. Deadlock may happen. We can see that T_1 have R-lock(A) and T_2 have R-lock(B). Then T_2 want W-lock(A) so it will wait for the unlock(A) from T_1. However, T_1 want W-lock(B), so T_1 will wait for unlock(B). Thus, both T_1 and T_2 cannot proceed, and encounter deadlock.

(2) By changing interleaving schedule, as follows:

T_1	T_2
read(A)	
read(B)	
read(C)	
write(B)	
write(A)	
	read(B)
	read(A)
	read(C)
	write(D)
	write(A)

By changing the lock mode:
We use a new "upgrade lock" mode (U-lock), which are actually read locks not compatible to themselves. This lock is used when the original lock is read lock but later it will acquire write lock. By using this lock, we can use U-lock(B) and U-lock(C) in T_1 in our problem. Then when T_2 try to acquire the U-lock, it will be blocked.

Q6 (15 marks) The "transfer" transactions T and U are defined as:

T: a.withdraw(40); b.deposit(40);
U: c.withdraw(30); b.deposit(30);

Suppose that they are structured as pairs of nested transactions:

T_1: a.withdraw(40); T_2: b.deposit(40);
U_1: c.withdraw(30); U_2: b.deposit(30);

We would like to consider the recovery aspects of the nested transactions. Assume that a *withdraw* transaction will abort if the account will be overdrawn. In this case, the parent transaction will also abort. Describe serially equivalent interleavings of T_1, T_2, U_1 and U_2 with the following properties:
1) That are strict
2) That are not strict

Based on your solutions, answer to what extend does the criterion of *strictness* reduce the potential concurrency gain of nested transactions?

Answer:

If a child transaction's abort can cause the parent to abort, with the effect that the other children abort, then strict executions must delay reads and writes until all the relations (siblings and ancestors) of transactions that have previously written the same objects are either committed or aborted. Our deposit and withdraw operations read and then write the balances.

For strict executions serially equivalent to T1; T2; U1; U2, we note that T2 has written B. We then delay U2's deposit until after the commit of T2 and its sibling T1. The following is an example of such an interleaving:

T_1: a.withdraw(40)	T_2: b.deposit(40)	U_1: c.withdraw(30)	U_2: b.deposit(30)
a.withdraw(40)			
	b.deposit(40)		
		c.withdraw(30)	
	commit		
commit			
			b.deposit(30)

The criterion of strictness does not in any way reduce the possible concurrency between siblings (e.g., $T_1$and T_2). It does make unrelated transactions wait for entire families to commit instead of single members with which it is in conflict over access to a data item.