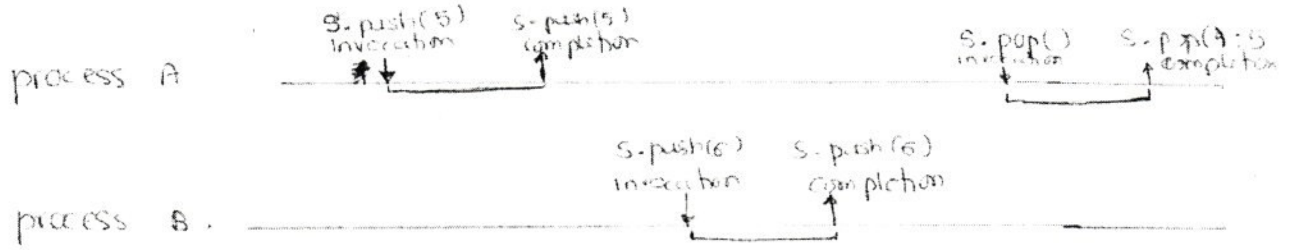
CHALMERS
EXAMINATION / TENTAMEN

Course code/kurskod	Course name/kursnamn			
TDA297	Distributed System Advanced Course			
Anonymous code Anonym kod		Examination date Tentamensdatum	Number of pages Antal blad	Grade Betyg
TDA297-9		2015-08-21	14	5

Solved task Behandlade uppgifter No/nr		Points per task Poäng på uppgiften	Observe: Areas with bold contour are to completed by the teacher. Anmärkning: Rutor inom bred kontur ifylles av lärare.
1	✓	10	
2	✓	10	
3	✓	12	
4	✓	5	
5	✓	8	
6	✓	10	
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
Total examination points Summa poäng på tentamen		55	

1.

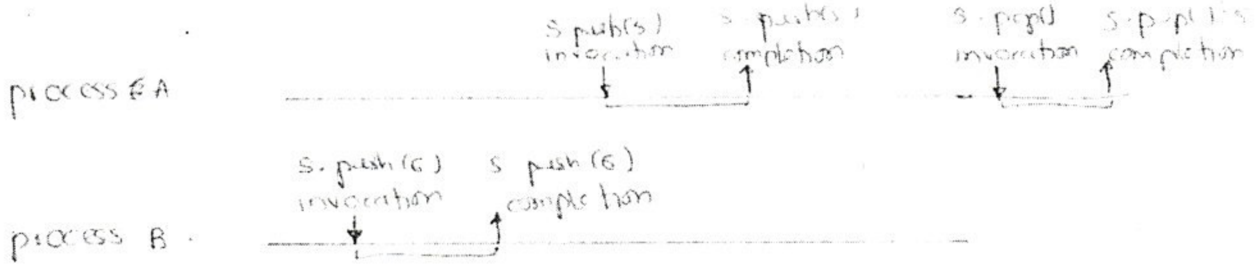
Let us assume that the concurrent stack behaves as a LIFO - Last in first out stack. We can illustrates the execution as follows. Stack is denoted by S



(a). This execution is not linearizable. Since the stack is LIFO the pop operation should pop 6. But it has popped (5). Because of this operation we cannot find an interleaving operations such that it ~~satis~~ meets the specification of a correct copy of LIFO stack with real time ordering.

5

- This execution is sequentially consistent. We can give the following interleaving that is sequentially consistent.



(b)

A shared replicated object service is linearizable if for any execution we can provide a some interleaving of operations such that

- The interleaving of the operations meets the specification of a single correct copy of the object.
- the order of the interleaved sequence is consistent with the realtime order where each operation occured.

5

A shared replicated object service is sequentially consistent if for any execution we can provide a some interleaving of operations such that

- the interleaving of the operations meets the specification of a single correct copy of the object.
- the order of the interleaved sequence is consistent with the program order where

1

each process executed from.

The difference between linearizability and sequential consistency is that linearizability requires an equivalent execution to follow the real-time order where as sequential~~ly~~ consistency requires an equivalent execution to follow the program order. Therefore in sequential consistency we can shuffle the order of operations in any combination as long as it gives a consistent correct view and operations from the same process are not shuffled.

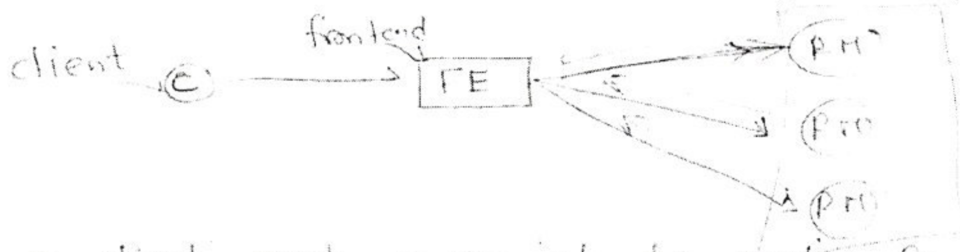
Therefore linearizability is a strict form of consistency and sequential consistency is a weaker form of consistency. Because of the real-time order linearizability also implies sequential consistency.

2.

~~As~~ We assume that our system provides a reliable, totally ordered multicast operation. Using this atomic broadcast we can design a replication based on the state machine approach that guarantees all replicas go ~~#~~ exactly through the same state transitions.

We assume
~~all replica~~
- each replica manager is a state machine
- each replica manager starts with the same initial state
- each replica manager (RM) receives same set of operations in the same order, so each RM can take identical but independent transitions.

We assume we have the following set up.



a client sends a request to ~~a~~ the front end FE and ~~FE~~ then FE multicast it to the group of RMs. The ~~steps can~~ precise steps are as follows.

- client sends its request to the FE.
- after receiving a request, the FE uniquely tag the request and multicast it to the group of replica managers.
- since we have a reliable, totally ordered multi~~broad~~cast every RM is going to receive the same set of messages in the same order. Also since each RM ~~ste~~ has the same initial state, now each RM can carry out each request independently. ~~but~~ After executing each step every ~~a~~ RM ends up in same state. Also each RM can send the corresponding response (with the id from request) back to the ~~replica manager.~~ FE. Additionally a RM can store the response so that if a duplicate request comes in a RM can send back the stored response without executing it.
- Once the FE receive the respond from one or more ~~rep~~ RM, it can choose to send a single selected response or ~~sy~~ single synthesized response back to the client.

2.

We assume the existence of the above described replication scheme. Then we consider a LIFO-stack with following operations

- ~~enq(x) – enqueue~~
- push(x) – push value x into the stack
- pop( ):x – pop a value from the stack - so it returns the last value x pushed ~~er~~ into the stack.

The LIFO-stack is implemented as an object ~~written on a file such that the file is replicated on each RM~~ (using a relevant data structure) and is copied on each RM. Additionally at the begining each ~~object~~ LIFO stack object is empty.

When a request comes in the FE multicast it to the group of RM as described in the previous section.

- Additionally in a push operation each replica manager sends a boolean value true or false as the response back to the FE. ~~wh~~ which then forward it to the client. In a pop ~~I~~ operation the popped value is sent back to the ~~cl~~ FE which then forwards it back to the client.

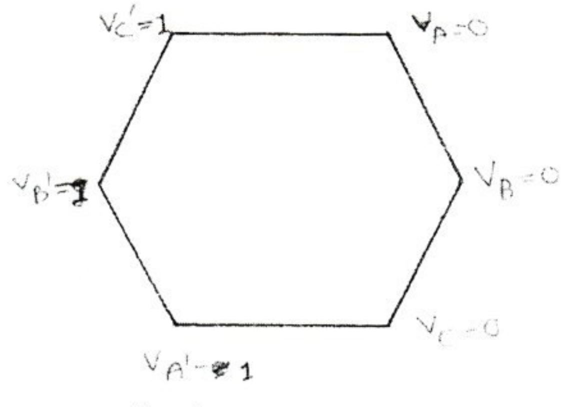
Since we have to guarantee high availability, when a FE gets a set of responses from RMs for a request, as soon as it gets the first response ~~th~~ FE sends it back to the client instead of ~~other~~ waiting for other ~~RS~~ RMs to respond. ~~I~~

Since each RM goes through exactly same state transitions choosing the first ~~acceptable~~ available response ~~also gives~~ is acceptable.

(a). In a distributed system if ~~f~~ a process behaves in an arbitrary non-deterministic way such type of faults are called Byzantine faults. In other words ~~for example~~ if a process shows different ~~dif~~ behaviours towards different processes they are called to be showing byzantine faults. For example a byzantine process can send different values to each of the processes in the group instead of sending the same value. Also ~~[illegible]~~ this type of faults include a processes taking arbitrary paths instead of the designated path.

(3)

(b). Assume towards contradiction there exists ~~an agr~~ a solution where three processes reach agreement when one of them is Byzantine faulty. Let this three processes be A, B, C. We also make copies of each of these processes and name them as A', B' and C'. Now we can arrange these processes as illustrated in ~~a~~ the following figure. Further we set the initial values of processes A, B, C to be $V_A = V_B = V_C = 0$ and values of processes A', B', C' to be $V_{A'} = V_{B'} = V_{C'} = 1$.



Now ~~for~~ each of the processes A, B and C sees that the are in the original three processor system. For example if we consider the connection $V_{C'} - V_A - V_B - V_C$ Processor A ~~sees~~ and B sees it is in a system with a third process C.

Let us consider the scenario S_0 with the connection $V_{C'} - V_A - V_B - V_C$. Now A and B see that it is in the original three process system where C behaving towards A as 1 and C behaving towards B as 0, with $V_A = V_B = 0$. In this scenario A and B can detect that C is faulty and decides 0 as the output agreement value.

Let us now consider the scenario S_1 with the connection $V_{A'} - V_{B'} - V_{C'} - V_A$. Now in this system processors B and C with initial value 1 sees that processor A behaving as 0 towards C and behaving towards B as 1. So B and C decides that A is faulty and agree on output value 1.

Lets consider a third scenario S_2 with the connection $V_B' - V_C' - V_A - V_B$. Now process A cannot distinguish this S_2 from S_C. So it decides 0 as the output value. On the other hand process C cannot distinguish this scenario S_2 from S_1. So it decides 1 as the output value. Now we have come to a situation where each process decides a different value and have not reach agreement. So this is a contradiction to our initial assumption.

(14)

There we have proven that it is impossible to reach agreement in a system with three processes if one of them is Byzantine faulty.

(C). We can generalize the ~~system~~ above proof for a system with n processes as follows.

First we assume that exists a protocol which solves agreement problem for a system of n processors when $n <= 3f$ and $f >= 2$. 'f' denotes the number of Byzantine faulty processes then our assumed solution worked for a system with at most 3f processors.
Then we divide the n processors into three different sets such that

$$1 \leq |A|, |B|, |C| \leq f.$$

(4)

Then we consider three processors P_A, ~~B~~ P_B and P_C such that each of them simulate the behaviour of the processors contained in sets A, B, C respectively.

We also assume that when we consider one set each process inside that set has the same initial value as the processor that represent ~~for the~~ the set. For example if P_A has 0 as the initial value then all the processors inside P_A has the same initial value as P_A. This is true for B and C as well.

Also the interaction between processors inside a set are simulated and interaction between ~~of~~ different sets are explicitly sent.

Now if one process inside a set ~~shown by~~ becomes Byzantine faulty, then at most f processors can become Byzantine faulty due to the way we divided the processes. But since we ~~assumed~~ initially assumed that this solution works for a system with $n <= 3f$ with $f \geq 2$, now we can conclude that it works for a system with three processors where one of them is Byzantine faulty. This is true because our simulation is a three processor system.

But this contradicts our selection from (b). Therefore we can conclude that our initial assumption is false.

Hence we have shown that there cannot be a n processor system where $n <= 3f$ and $f \geq 2$ that achieves agreement. Therefore we need ~~→~~ n to ~~be~~ be $n \geq 3f+1$.

(d). Yes, it is possible to reach agreement in a system with three processes if one of them is Byzantine faulty by using authentication.

We declare the following assumptions for the algorithm.

- any process can ~~de~~ identify which process ~~send~~ sent it a message.

(4)

- a loyal general's/lieutenant's signature cannot be forged. Any alterations to their signed messages can be detected.

- a loyal general/lieutenant can verify the signature of ~~and~~ other loyal general/lieutenants.

We also assume a function called ~~Choice~~ Choice (V) defined as follows.

V is a proper set which contains no duplicate values. It can have only either/both R (=retreat) and A (=attack).

- when $V = \phi$ (empty) then choice (V) = R
- when $V = v$ ($v = A$ or $v = R$) then choice (V) = v
- when $V = \{R, A\}$ then choice (V) = R

The notation $v:i$ denotes the decision v of general/lieutenant signed by himself.

$v:i:j$ denotes the message $v:i$ counter signed by general/lieutenant j.

The algorithm is as follows. The general always has number 0.

- On initialization the General send$(v:0)$ to all lieutenants.

- Each lieutenant L_i executes the following
  - initially the set V is empty.
  - upon receiving a message $(v:0:i_1 \ldots i_k)$ it tries to validate all the signatures. If it passes then it add decision v to V
    $V = V \cup v$.
  - then it checks if $k < f$ (i.e. if ~~the~~ don't have enough signs).

    Then for each j in $\{1 \ldots n-1\}$ & $i_1 \ldots i_k$ send $(v:0:i_1 \ldots i_k)$ for each L_j

  after above is done and if L_j come to state where it is not going to receive any more messages then the agreed value is
  choice(V).

In this algorithm where $n \geq f+2$

- if the general is correct then after first round everybody choose general's value.
- if the general is Byzantine faulty after f rounds all lietenants can see that general is Byzantine faulty and choose to retreat.

~~5~~ 4

(i) Reliable broadcast

A broadcast is said to be reliable if it is satisfies the properties: integrity, validity and agreement. Each of these properties are defined as follows

Integrity
- A correct process 'p' delivers a message ~~at~~ 'm' at most once and only if some other process in the same broadcast domain has broadcast it.

Validity
- If a correct process 'p' ~~deliver~~ broadcast a message 'm' then it will eventually deliver that message 'm'.

Agreement
- If a correct process 'p' delivers a message 'm' then all other correct processes in that group will eventually deliver that message 'm'.

(ii) FIFO broadcast

If a correct process broadcast a message 'm1' and then a message 'm2', all the correct processes that deliver m2 delivers m1 before m2.

(iii) Causal broadcast.

If there are two broadcast messages m1 and m2 such that they are related with m1 → m2 where '→' defines m1 happened before m2, then any correct process that delivers m2 delivers m1 before m2.

5

5

We consider in the network G(V,E) there is one process that initiates the spanning tree computation and other processes participates in the calculation.

The algorithm is as follows.

For initiator {

N = {q | q is a child neighbour of the initiator}

for each q
send token for start tree construction

ACK = N

while (ACK ≠ ∅){ empty
upon receiving an ~~ACK~~ acknowledgement from a child process ~~q~~ ~~a child~~ q
ACK = ACK - {q}
}

after ACK == ∅
terminate.

}

8

For a other process p :{

receive the token from the parent for tree construction

N = {q | q is a child neighbour of p}.
for each q
send token

ACK = N
while (ACK ≠ ∅) {
upon receiving an acknowledgement ~~from~~ from a child process q
ACK = ACK - {q}.
}
after ACK == ∅

send acknowledgment to parent

terminate.

}

ACK is a set that is maintained by each process which contains all the child neighbours it has sent token for spanning tree construction. So that when the parent ~~receives~~ can keep track of which processes have completed its part of spanning tree.

5

In this algorithm,

- when a process q receives a token from another process p ~~it~~ ~~mark~~ for the first time q marks p as its parent. ~~p sends ackn~~
- q sends acknowledgement back to p only after q receives acknowlegements from all its children.

- when constructing the tree the first edge where a process q receives ~~from~~ a token, is marked as ~~the~~ a edge of the spanning tree so that all the first-used edges will form the spanning tree.

- when a process q has ~~its~~ already decided its parent, if it receives the same token from another process q just sent back an acknowledgement but does not change the parent.

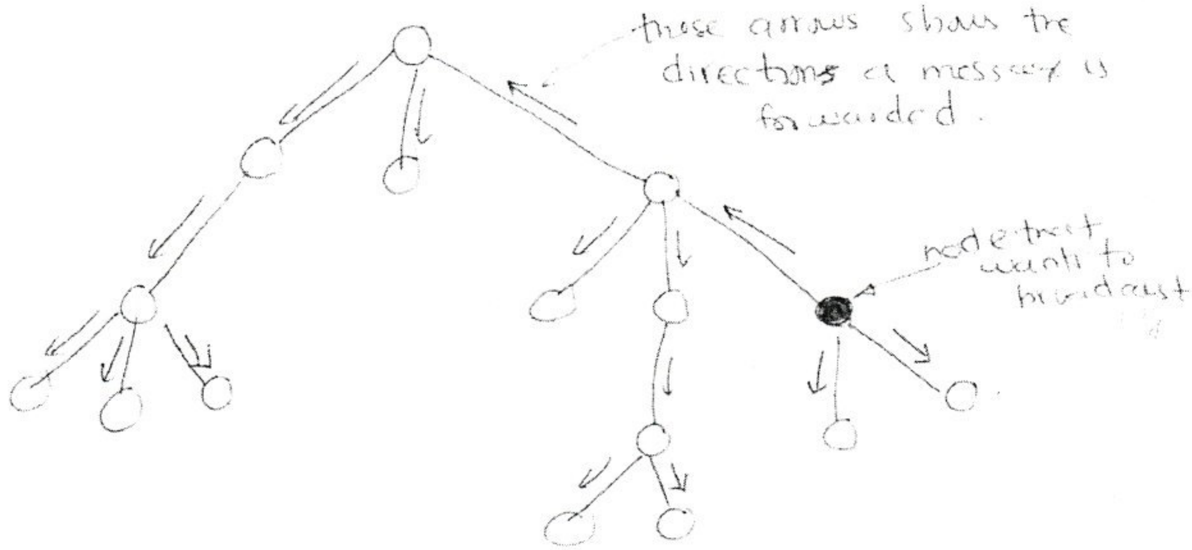
The existence of such a spanning tree can be used as follows in order to perform broadcasting

- If the process that wants to broadcast is the root node it can simply send the message to all its child nodes. Then each child node will ~~forward~~ forward this message to its child nodes. ~~By recurs~~ Recursively ~~all the~~ following this process each ~~at each~~ node of the tree is going to receive the message.

- If the process that wants to broadcast is a node other than the root node, then it can do the following things
  - it sends the message to all its child neighbours which will eventually ~~sends~~ forward it to each of ~~their~~ their child neighbours. So that the origin node can cover that part of the subtree.
  - in order to send it to other nodes which are not in its subtree it can forward this message to its parent node then that parent node can forward this message all of its child nodes (except the child that send the message. ~~At~~ So that parent can cover its part of the subtree.

~~It~~ - Also if there is a parent to that parent it ~~has~~ the first parent has to send the message to its parent.

Likewise this can continue recursively until all the nodes are covered. This ~~e~~ second case can be ~~shown~~ shown as follows using the following figure.



Yes this is a good solution that can be used on a wireless sensor network, with battery constraints.

Because, first it calculates the spanning tree in $O(n)$ ~~with $O(\delta$~~ time with $O(\delta)$ message complexity
n - # of nodes   δ - degree

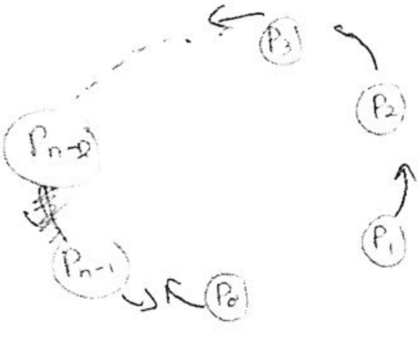
After that ~~time complexit~~ since we are using the same spanning tree the time complexity can be reduced to $O(\Delta)$ where the Δ denotes the depth of the tree though message complexity remains same.

The only drawback is in the first phase when calculating the tree since we use acknowledgement each edge has to transmit two messages where as after that each edge transmit only one message.

6

2 Yes this solution solve the dinning philosopher problem.

In order to prove this fact and time complexity we assume the following configuration with n processes



We also assume that each fork has a associated FIFO queue. So that only the process in top of the queue can access the fork.

Let us assume that a process take K>>time units to eat after it has obtained its two forks.

If we consider ~~a pr~~ the process p_0, the worst case is that when process p_0 tries to get its first fork all the other processes P_1 to P_{n-1} have grab their first fork. This means that when P_0 tries to get its left fork P_{n-1} has already grabbed that fork. But inorder P_{n-1} to eat it has grab its second fork which is its left fork. But ~~it he~~ that fork has already been grabbed by P_{n-2}. By induction we can show that the only process that can eat this stage is P_1. So it eats for K time units and releases its two forks. Now P_0 has its second fork. But since it has not obtained its first fork it cannot grab the second fork. This means that in order P_0 to grab first fork all the processes P_1 to P_{n-1} have to complete eating. So this will take $t_1=(n-1)K$ time units.

After t_1 time units P_0 can get its first fork. Then when it can compete for second fork shared with P_1. But in ~~a~~ the worst case P_0 has to wait another k units ~~a~~ if p_1 has already grabbed that fork.

So altogether P_0 has to wait ~~(n-1)k + k = n~~

$$(n-1)k + k = nk \text{ time units.}$$

So for P_0 time complexity is nk.

For processor P_1 this is $2k$ time units because first fork to P_1 is second fork to P_2. So k units have to be waited to get first fork. Then second fork to P_1 is second fork to P_0. if has ~~that~~ this fork that means ~~it has~~ P_0 has got its first fork as well. So P_1 waits another k units until P_0 is done.

14

6

If we consider another node which is not P_0 & P_1

In the worst case it has to wait less than nk time units Because when it competes for the first fork if its neighbour has grabbed it it has to wait k units until P_{j+1} is finished. Then when it ~~waits~~ ~~for~~ competes for the second fork in the worst case it has to wait until all process P_1 to P_{j-1} finish eating so this yields $k \times (j-1)$ units altogether a process P_j has to wait $k(j-1)+k$ $=$ ~~kj~~ $= kj$ time units which is less than nk.

$jk < nk$ because $j < k$.

In order to prove that ~~we~~ this solution to solve the dining philosophers problem we have to show that it guarantees mutual exclusion and no starvation.

~~At the start~~

No starvation

- By the above complexity calculation we have already prove that there is no starvation. Because ~~for~~ P_1 - ~~it tak~~ succeeds in $2k$ units; P_0 - succeeds in nk units; any other P_j ($j \neq 0,1$) succeeds in jk units.

Mutual exclusion

- For process P_0, ~~the first fork for it is the~~ its first fork is left fork. For process P_{n-1} its first fork is its right fork. This means for both P_0 and P_{n-1} the fork between them is the first fork.
  In the same manner the fork between P_0 and P_1 is the second fork for both P_0 and P_1. Since we use a FIFO queue only the process on the top of the queue can access the fork of those two forks. So mutual exclusion is guaranteed.

In other cases where we have $P_1, P_2, \ldots P_{n-1}$ the fork between each consecutive pairs P_1-P_2, P_2-P_3, ..., $P_{n-2}-P_{n-1}$, is going to be the first fork for P_j and second fork for P_{j+1} ($j \neq 0$). But again since we have a FIFO queue only once process succeeds in getting the fork.

Additionally this configuration does not create any deadlocks since P_0 breaks the symmetry. So we also have the progress.