Concurrence, Parallelism and Distributed Systems (CPDS)
Module III: Distributed Systems
Facultat d'Informàtica de Barcelona
Final Exam
June 19th 2018

Answer the questions concisely and precisely
Answer in the same sheet
Closed-book exam
Duration: 2 hours

Name and Surname:

---

1. (5 pts) Select the answer (only one) that you consider to be correct on every section. Every correct answer computes 1/2 points. Every wrong answer deducts 1/6.
   (a) Given a system to check the current weather, which of the following types of failure occurs if two users in the same location check the weather at the same time and get different responses?
      - Arbitrary (Byzantine) failure
      - Crash failure
      - Timing failure
      - Omission failure

   (b) Which of the following is an indirect communication paradigm?
      - Remote invocation
      - Message passing
      - Stream-oriented
      - Group communication

   (c) Which of the following statements about the Cristian's algorithm to synchronize physical clocks is false?
      - Each client asks the time to the server at every resynchronization interval
      - Each client sets its time to $T_S + RTT$, being T_S the time within the message received from the server and RTT the round-trip time (i.e. the elapsed time between the client's request and the server's response)
      - Accuracy of client's clock is $\pm(RTT/2 - Min)$, being RTT the round-trip time and Min the minimum latency between the client and the server
      - The resynchronization interval must be lower than $\delta/2\rho$, being δ the maximum allowed clock skew and ρ the clock drift

   (d) Can we obtain totally-ordered logical clocks by using Lamport's clocks?
      - No, vector logical clocks are needed to obtain a totally-ordered sequence of events
      - No, Lamport's clocks define a partially-ordered sequence of events and there is not way to extend them to define a totally-ordered one
      - Yes, despite Lamport's clocks define a partially-ordered sequence of events, they can be combined with the process identifier, which will be used to break the tie in case of two clocks being equal
      - Yes, Lamport's clocks already define a totally-ordered sequence of events

(e) Let a be the sending of a message and b be the reception of that message, which of the following statements is true?

- There may be some linearizations of the execution where b does not appear
- There may be some linearizations of the execution where neither a nor b appear
- Both a and b will appear on any linearization of the execution, but they could appear on any order (a before b or b before a)
- Both a and b will appear on any linearization of the execution, and a will always appear before b

(f) Which of the following mutual exclusion algorithms has the lowest synchronization delay (i.e. number of message latencies between a process releasing the critical section and the waiting process entering it)?

- Centralized algorithm
- Lin's algorithm
- Ricart and Agrawala's algorithm
- Maekawa's algorithm

(g) The *Byzantine generals problem* deals with solving consensus in a scenario with ...

- Crash-faulty processes and reliable communication in a synchronous system
- Byzantine-faulty processes and reliable communication in a synchronous system
- Faulty processes and reliable communication in an asynchronous system
- Non-faulty processes and unreliable communication

(h) Which of the following statements about the *Dolev and Strong algorithm* to solve consensus is false?

- It proceeds in $f + 1$ rounds, being f the maximum number of processes that may crash
- On each round, each process multicasts the set of values that it has not sent in previous rounds
- On each round, each process collects the values multicasted by other processes and records any new values
- A round does not terminate until the values from all processes are collected, waiting for crashed processes to recover if needed

(i) Which of the following statements about *strict two-phase locking* concurrency control for distributed transactions is false?

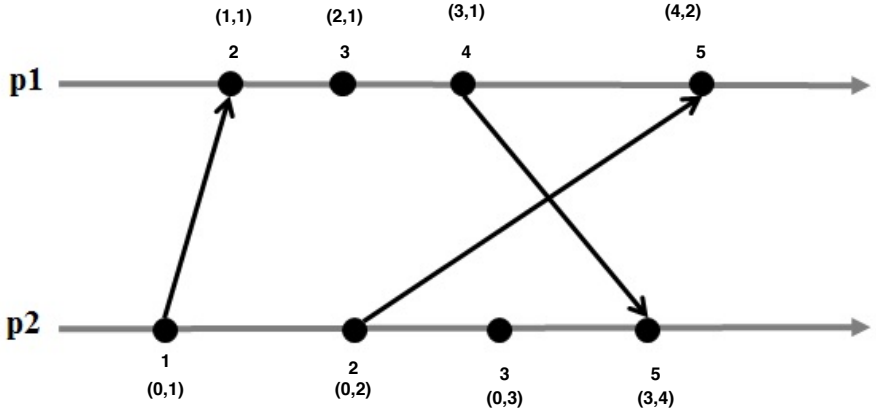
- Each server maintains locally the locks for its own objects
- A transaction cannot acquire new locks after it has released a lock
- Different servers will order in the same way the transactions accessing their objects
- Locks are released all at once when the transaction commits or aborts at all the servers

(j) What is the main characteristic of *strong consistency models*?

- They provide a system-wide consistent view of the data with a granularity of individual operations on data stores with concurrent writes
- They provide a system-wide consistent view of the data with a granularity of individual operations on data stores without concurrent writes
- They provide a system-wide consistent view of the data with a granularity of groups of operations on data stores with concurrent writes
- They provide a consistent view of the data for each client with a granularity of individual operations

Name and Surname:

2. (1 pts) Given the following events performed by P1 and P2:

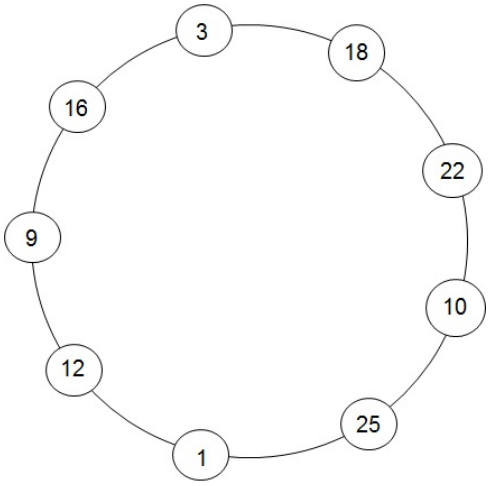


a) Tag each event with its corresponding Lamport's logical clock.

b) Tag each event with its corresponding vector logical clock.

c) Draw the lattice with the reachability relation between the resulting consistent global states.

3. (1 pts) Given the following set of processes organized in a logical unidirectional ring:



a) If the processes 12 and 18 initiate two concurrent elections to choose a coordinator, indicate how many messages (and which ones) will be sent for each election until both have finalized in the following two cases:

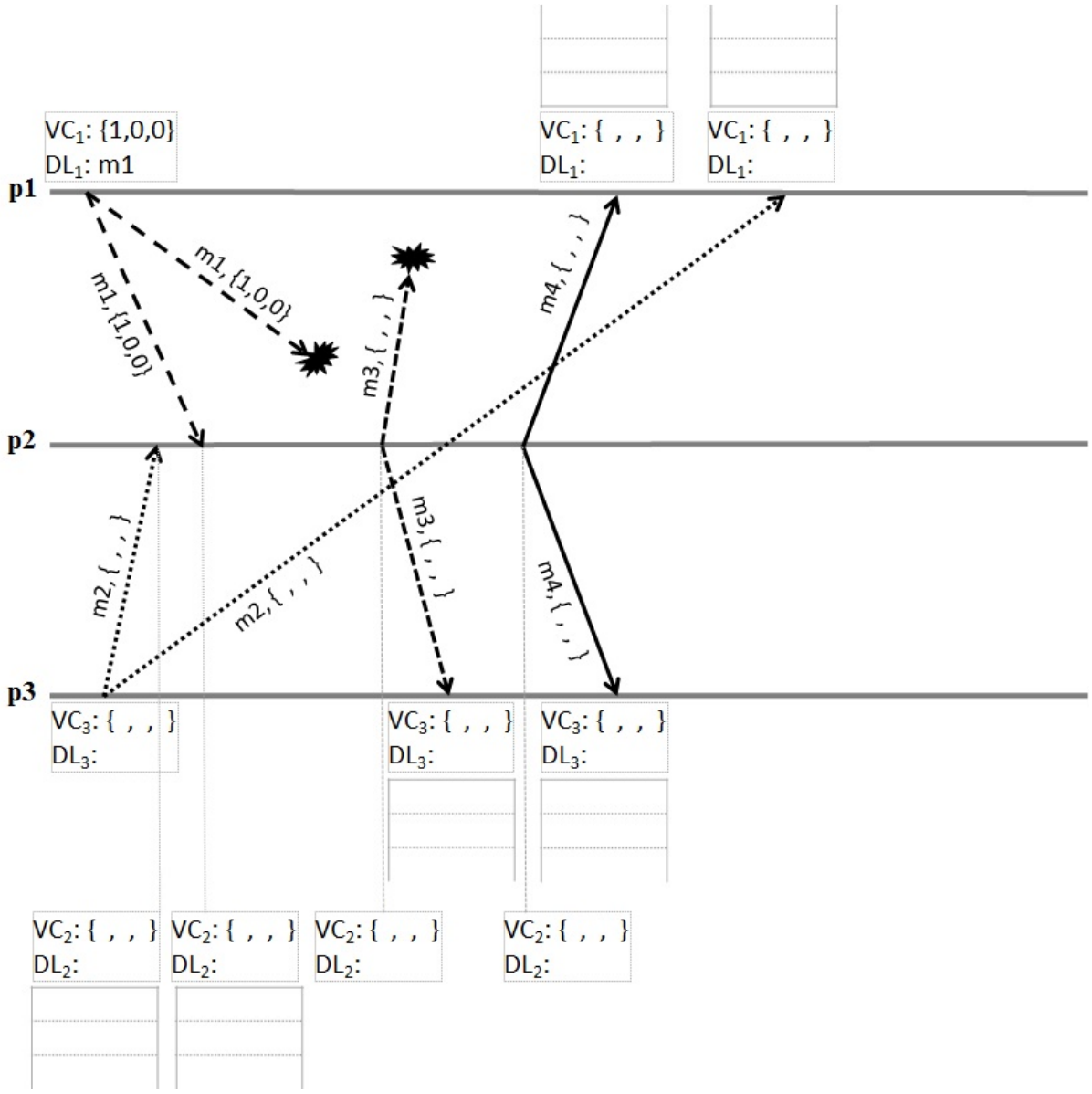
i) Processes use the *Chang & Roberts' algorithm.*

ii) Processes use the *enhanced ring algorithm.*

b) Explain how the *enhanced ring algorithm* could be modified to remove the redundant election messages that occur when there are several concurrent elections ongoing, as done by *Chang & Roberts' algorithm* (hint: use the identifier of the process that initiates the election to decide which one must have preference).

Name and Surname:

4. (1 pts) Given a group of three processes that communicate through *causally-ordered scalable reliable multicast*, complete the following figure indicating for each process i: its vector clock (VC_i), the messages stored at the hold-back queue pending to be delivered, the list of messages delivered up to now (DL_i), and the vector clock of sent messages. *Scalable reliable multicast* is implemented by assuming that a process that queues a message in the hold-back queue because it has not received yet the messages that casually precede it, will request the retransmission of those missing messages by sending the corresponding NACKs. If the process has already requested the retransmission of some message before, it will not do it again. Draw also in the figure the sent NACKs and the retransmitted multicast messages, and indicate also how each receiver process deals with each retransmission (recall that duplicated messages have to be discarded). For delayed messages (but not lost), you can assume that the retransmission arrives after the original delayed message.



5. (1 pts) We have two transactions operating concurrently on the same objects in a distributed system implementing *timestamp ordering concurrency control*. The initial values of the objects i and j are 11 and 22, respectively, and their initial read and write timestamps are t_0. Indicate the effects of each operation performed by transactions T and U in the two interleavings displayed in the following figures. For each operation, you have to state the validation performed to decide whether the operation may proceed (and its outcome) and the timestamps and values of committed and tentative objects. Note that the first transaction getting a timestamp will get timestamp t_1 and the second will get timestamp t_2, where $t_0 < t_1 < t_2$.

1)

T	*U*
openTransaction	*openTransaction*
x= read (i);	*write(i, 55);*
write(j, 44);	*x= read (j);*
commit	*commit*

2)

T	*U*
openTransaction	*openTransaction*
x= read (i); **or** */ write(j, 44);*	*write(i, 55);*
commit	*x= read (j);*
	commit

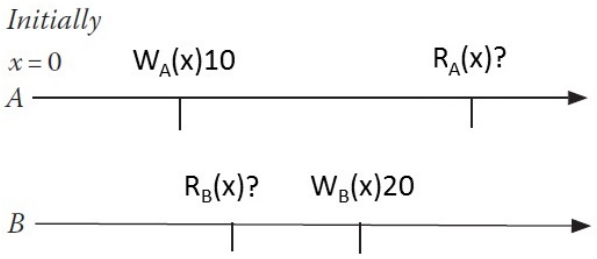
- Figure 1)
  - `U: write(i,55):`
  - `T: x=read(i):`
  - `T: write(j,44):`
  - `U: x=read(j):`
  - `T: commit:`
  - `U: commit:`
- Figure 2)
  - `U: write(i,55):`
  - `U: x=read(j):`
  - `U: commit:`
  - *Option a)*: `T: x=read(i):`
  - *Option b)*: `T: write(j,44):`
  - `T: commit:`

Name and Surname:

6. (1 pts) Given a data store consisting of 6 replicas that uses a *quorum-based replicated-write protocol* where $N_R = 2$ i $N_W = 4$.

   a) Given the previous values of N_R and N_W, justify whether there are *read-write* and *write-write* conflicts.

   b) Explain the actions that will take place (what replicas are accessed, what operation is performed in them, what happens with the rest of replicas) if clients A and B execute the operations as shown in the following picture, assuming that the operations obtain the next quorums: $N_{W_A} = N_{W_B} = \{1, 3, 5, 6\}$, $N_{R_B} = \{4, 5\}$, $N_{R_A} = \{2, 4\}$.



   c) What value will be returned by operations $R_A(x)$? and $R_B(x)$? ? Justify your answer.

   d) If $R_A(x)10$ and $R_B(x)10$, what consistency model is this data store providing?

7. (SEMINARS) GEQ. **Paxy**.

a) (6 pts) Complete the following code excerpt corresponding to the *acceptor* process:

```
acceptor(Name, Promised, Voted, Value) ->
  receive
    {prepare, Proposer, Round} ->
        case order:goe(  ...  ,  ...  ) of
            true ->
                  ...  ! {promise,  ...  ,  ...  ,  ...  },
                acceptor(Name,  ...  ,  ...  ,  ...  );
            false ->
                  ...  ! {sorry, {prepare,  ...  }},
                acceptor(Name,  ...  , Voted, Value)
        end;
  end.
```

b) (2.5 pts) Make the needed adaptations in the previous code to enable persistence of the acceptor's state to a file named *Name*. You can use the functions from the *pers* module (open, read, store, close, delete). Justify why you call the functions of the *pers* module in those places (and only on those places).

c) (1.5 pts) Reason what the impact is when increasing the number of proposers while keeping the same number of acceptors.

Name and Surname:

8. (SEMINARS) IEQ. **Opty**.

a) (5 pts) Complete the following code excerpt corresponding to the *validator* process:

```
validator() ->
    receive
        {validate, Ref, Reads, Writes, Client} ->
            Tag = make_ref(),
            Self = self(),
            lists:foreach(fun({Entry, Time}) ->
                  ...  ! {check, Tag,  ...  , Self}
                end,  ...  ),
            case check_reads(length(  ...  ), Tag) of
                ok ->
                    lists:foreach(fun({_, Entry, Value}) ->
                          ...  ! {write,  ...  }
                        end,  ...  ),
                    Client ! {Ref,  ...  };
                abort ->
                      ...  ! {Ref,  ...  }
            end,
            validator()
    end.
```

b) (2.5 pts) Reason what the impact on the percentage of committed transactions with respect to the total is when ...

i) we increase the number of write operations per transaction.

ii) all the transactions perform only write operations.

c) (2.5 pts) Given the following start-up code which is executed in an Erlang instance named `opty-1` and running in machine `cpds.fib.upc.edu`:

```
start(Clients, Entries, Reads, Writes, Time) ->
    spawn('opty-2@cpds.fib.upc.edu', fun() -> register(s, server:start(Entries)) end),
    startClients(Clients, [], Entries, Reads, Writes).

startClients(0,L,_,_,_) -> L;
startClients(Clients, L, Entries, Reads, Writes) ->
    Pid = client:start(Clients, Entries, Reads, Writes, s),
    startClients(Clients-1, [Pid|L], Entries, Reads, Writes).
```

i) Say where the client, server, validator and handler processes will be executed.

ii) Fix the code to enable a correct communication between the clients and the server running in different Erlang instances.