1. a => b, c => d, b => d, a=>d ;
   1. By the first condition of Lamport’s happened-before relationship, we know that a => b since they are on the same process and a was executed before b (by the timestamp). By the same logic, c => d, so we know that c happened before d. By the second condition of the happened-before relationship, we know that b => d since it is given to us that b sends a message to d. By the third condition, we know that a => d since we already know that a => b and b => d. So officially, we know that **a, b, and c happened before d**.
      1. **Moreover, we don’t know anything about which events happened before e since none of the other events are connected to e.**
2. Here are the final values of each Lamport clock:
   1. a : 1
   2. b : 2
   3. c : 1
   4. d : 2
   5. e : 1
3. There is no exactly once semantics because we cannot guarantee that a message will successfully be sent and received in the face of network failures. We can work around this limitation by choosing between at most once/at least once semantics. ?
4. While the messages are in the middleware waiting to be delivered to the application, the messages should rely on the timestamps and their orderings in order to determine when to be delivered. If a message is delivered before its immediate predecessor (in terms of timestamps), then it has been delivered “too early”. We can avoid this by enforcing this rule: “the current message has to wait until the message immediately before (based on timestamp) it is delivered before it can be delivered itself”.
   1. The relationship between the timing of the delivery and consistency is that the more consistent a system is, the more time it takes to deliver the messages, since each message will have to wait for its predecessor to be delivered first. On the other hand, the more available a system is, the less time it takes, since messages can be delivered as soon as they are received in the middleware, but this introduces the possibility of delivering these message “too early”.
5. (a). In the case of fail-stop semantics, part of the server has broken down, i.e. maybe the computer’s harddrive failed. In this case, the RPC should flag that server as having problems, so the server can take appropriate action to resolve that issue and the client can account for the situation if the server completely fails.
   1. (b). In the case of fail-crash semantics, the whole server has failed, so the RPC should flag that server as failing and should stop sending/receiving data from that server (close the connection).