Victor Chu
October 6, 2016

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**.

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:

a	1
b	2
c	1
d	3
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 and errors. We can work around this limitation by choosing at-least-once semantics and allowing our messages to sent many times until there is exactly one message received. We can only do this if sending many messages has the same effects as sending exactly one message, i.e. there are no side effects.

4. While the messages are in the middleware waiting to be delivered to the application, the messages should rely on its own timestamp as well as its immediate predecessor's timestamp. 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".

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 hard drive failed. In this case, the server should flag itself as having problems so the RPC will know, and the RPC should stop

sending/receiving data from that server temporarily until the issue is resolved and when the server unflags itself.

(b). In the case of fail-crash semantics, the whole server has failed, so the RPC connection will crash. In this case, the RPC doesn’t know what happened to the server, so should send out heartbeat signals every so often in order to verify the server’s status. When the heartbeat does not come back from the server, since it crashed, the RPC should terminate the connection until the server comes back up, if it ever does.