Fabien Bessez
Homework 3
Distributed Systems

1. **The following events took place on three systems at the stated physical (time of day) times: Event b is the sending of a message from system 1 to 2. Event d is the receipt of that message. Using Lamport's happened-before relation, which events happened before event d? Which events happened before event e?**

    i. **• System 1:– a–3:00 – b–3:01**

    ii. **• System 2:– c–2:55 – d–3:04**

    iii. **• System 3:– e–3:03**

    Lamport's *happened-before* relation has three rules: 1) If A and B are actions on the same process, and A was executed before B, then we can know that A => B. 2) If A is a send and B is a receive on a different process , then A => B. 3) If A => B and B => C, then we know that A => C. Now, to talk about which events we know happened before event d: By rule 2, we know that event B => D. By rule 1, we know that A => B because it is on the same process. By rule 1, we also know that C => D. We also know by rule 3, that A => D. But we do NOT know if there exists a happened-before relationship between A and C. Which events happened before event e? We do not know since system 3 is never communicated with!

2. **Using the example system in the previous question, assume that all Lamport clocks are initialized to zero. What is the final value of each Lamport clock?**

    I believe the final values of each system are as follows: System1 = 0 -> 1 -> 2; System2 = 0 -> 1 -> 3; System3 = 0 -> 1.

3. **We discussed at-most-once semantics and at-least-once semantics of message delivery. Why is there no exactly once semantics, and how can we work around this limitation?**

    In exactly once-semantics, if there is a partition tolerance issue, there has to be some sort of guarantee that the message, that was sent exactly once, still makes its way to the server. One could imagine a scenario when Client1 sends a message to Server1 but after being sent and before arriving at Server1, Server1 crashes! To get around this, we would have to implement a way to keep track of the messages that are "in the tube". A solution could be an intermediary server (Server2) that runs on an at-least-once semantics. In this scenario, Client1 sends the message to Server2, which then continues to send the message to Server1 until it is received. A hypothetical

work around this limitation is to ensure partition-tolerance (but that's just hypothetical).

4. **We talked about how a system can use Lamport clocks for ordering messages to be delivered to an application. Imagine that a computer has received, but not yet delivered, some messages. It has sorted them according to their timestamps. What factors should be considered in deciding when to deliver the messages to the application? What would constitute "too early" delivery, and how can we avoid it? Discuss in particular of relationship of timing of the delivery in relation to the balance between consistency and availability.**

   Some factors to consider in deciding when to deliver the sorted messages after they've arrived are: Are there any missing messages? In other words, does the application have message 1, 2, and 3 or just 1 and 3? The application can tell this based on the Lamport timestamps. Are there any duplicates that need to be de-duplicated? How long has it been since the messages have arrived?

   What would constitute "too early" delivery, and how can we avoid it? A message is delivered "too early" if the Lamport timestamps indicate that there exists a message that should have arrived before the latest message you've received but you send the latest message anyway. In our Skype example, if you receive the "L" portion of "HELLO" before the "H" and the "E", it would be "too early" to send the "L". However, after a certain amount of time that depends on the application service, it would be wise to just send the "L" and forget about the missing "H" and "E". In the Skype example, it would be wise to do that. To avoid sending MessageB too early, the server could request that MessageA (that happened-before MessageB) be re-sent! It is important to note the trade-off present between consistency and availability. To remain consistent, it might be of value to stop accepting messages from the client once the server realizes it is missing a prior message. To remain available, the server might continue to receive messages but not deliver any until it has the correct ordering. Vector Clocks can be used to detect causal dependency and solve sending things "too early". They keep track of every process's individual Lamport clock timestamp in an array.

5. **Describe how an RPC system should respond to a server's failure in the case of (a) fail-stop semantics and (b) fail-crash semantics.**
   When an RPC system notices a server's failure under fail-stop semantics, the server will stop producing output in such a way that it's halting can be detected by other processes. It should be able to somehow notify the client that there is an

issue and that is no longer accepting requests. When an RPC system notices a server's failure under fail-crash semantics, the connection should be dropped and the client should be notified that there was an issue with either the connection or the server. Fail-crash is when the server crashes without notice to the client. In that case, it would make sense for the server to stop being available and for the client to just wait.