ESE 605 Homework 3

Stephen Phillips

February 28, 2015

Problem 1

Part a

Due to network conditions, the transmission time between one machine and another can vary quite a bit. Therefore by taking statistics of previous transmission times, we can lower the error of the clock skew measurement by getting a better measurement of the actual transmission time. We do this by sending multiple messages then taking an average or the median to determine the acutal transmission time. Then

Part b

It doesn't make a difference, since the results of both will give the process with the highest process ID the position of leadership. So the outcome of both elections will be the same and the final outcome will not change. There may be more wasted communication, but it won't affect how the system behaves.

Problem 2

Part a

Reading and Writing files from a file server are best done with **at least once** semantics. Reading a file clearly is idempotnent, and since we specify the exact positions in the file to write to, and the data to write to it, then the write becomes idempotent. Since these are both idempotnent operations, to guarantee their success, we can send the message as many times as we want to.

Part b

Compiling a program should be done with **at least once** semantics since the compiled code only depends on the source code files. Because of this, commands to compile the code can be sent more than once. While this may lead to unnecessary computation, it will guarantee that the code is compiled which is what we want more.

Part c

Online banking should be **at most once** since the transactions need to happen only once. Ideally we would use exactly once, but unfortunately, that cannot be implemented in the real world. So as we do not want to for example send money to an account twice, we use at most once semantics. We could make this to at least once if we were clever, using transaction IDs or something but barring complicated implementation you need to use at most once.

Problem 3

Part a

Lamport's logical clock gives the guarantee if $A \to B$ (B happens after knowledge of A) then $L(A) < L(B)$ (logical clock value of A less than the logical clock value of B). This is enforced by how Lamport's logical clock is run. Since every event within a process increments the logical clock, within a process this holds.

Between processes, since when a process gets a message it sets is clock to the max of its own clock and the message clock plus 1. Therefore it maintains this invariant. Concurrent events are not considered.

Part b

Since each process runs on its own logical clock then the reverse of part a, that $L(A) < L(B)$ means $A \to B$, is not true. For instance consider 3 processes P_1, P_2, and P_3. Let's say P_1 sends a message to P_2 and P_3 with time stamp 9. Then P_2 sends a message with time stamp 6 to the other processes. From these time stamps we can say nothing about which came first, whether one came before the other or they happened concurrently. So even though P_2 sent its message later it has the smaller timestamp.

Vector clocks fix this by maintaining logical clocks for each of the processes in the system. So each process knows the logical clock for all the other processes, and uses them to determine late messages. For instance say we have processes P_1, P_2, and P_3 again. This time they are maintaining vector clocks. Say that P_1 sends a message to the others, with time stamp say (2,0,0) and then P_2 gets it and sends another message with time stamp say (2,3,0). If P_3 gets the message from P_2 first, with time stamp (2,3,0), it knows it has missed a message from P_1 since it has a clock of something like (0,0,1). And therefore waits until it has recieved P_1's message with time stamp (2,0,0) before it sends any of its messages to the application that it is running.