University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

ECE 454
Assignment 5

Prepared by
Chan, Carl
UW Student ID Number: 20383063
UW User ID: c73chan@uwaterloo.ca
and
Li, Debin
UW Student ID Number: 20389489
UW User ID: d73li@uwaterloo.ca

7 July 2014

1. Proof: If every $C_i(\cdot)$, $C_j(\cdot)$ satisfy k_1 and k_2, the $C(\cdot)$ satisfies k_0

   In k_0, it states that $a \rightarrow b \; implies \; C(a) < C(b)$.

   In k_1, if $a \rightarrow b$ in P_i, the $C_i(a) < C_i(b)$. That means that any events originating within a process can be ordered using their timestamps.

   As stated in k_2, if a is the sending of a message at P_i, and b is receipt of that message at P_j, then $C_i(a) < C_j(b)$. This means that if P_i sends a message a to P_j, who then causes a message b to be sent back to P_i, then within P_i the condition $C_i(a) < \; C_j(a) < C_i(b) \rightarrow C_i(a) < C_i(b)$ holds by transitivity. This means that regardless of what happens outside a process, the timestamps from messages received can always be used to determine which internal events occurred before and after the message.

   If every C_i and C_j satisfies k_1 (can order internal events) and k_2 (can order external events relative to internal events), we can conclude that $C(\cdot)$ also satisfies k_0

2. Proof: $a \rightarrow b$ if and only if $VC(a) < VC(b)$

   We know by the definition of $VC(a) < VC(b)$ for two events a and b if for all i, $VC(a)[i] \leq VC(b)[i]$, and there exists j such that $VC(a)[j] < VC(b)[j]$. $a \rightarrow b$ means that a causally precedes b. What it means to causally precede is for something to must be before something else in time.

   This happens if the understanding of time in each of the involved processes says that the number of messages sent at the time a was sent is lower than the number of messages sent when b was sent. This is as all processes along the chain of causality will increment their vector clock for themselves whenever sending a message, so at least one element of the overall vector clock must always be less for the causally preceding message. Any process not involved in the causal relationship will either have their own understandings of time remain the same (which accounts for the equality), or be greater as time goes on (in which case the later message will have a greater VC value for these processes as well). This only happens if $a \rightarrow b$, as "concurrent" messages would have incremented at least one VC element which the other did not, causing the comparison to fail.

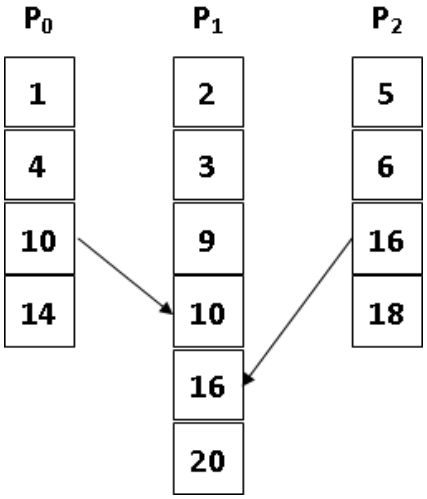
   This proves both that $a \rightarrow b$ implies $VC(a) < VC(b)$ and that $VC(a) < VC(b)$ implies $a \rightarrow b$. Therefore, $a \rightarrow b$ if and only if $VC(a) < VC(b)$.

3. Due to the fact that a→b has C(a) < C(b) in the protocol, any message which causally precedes another will be passed to the process in the correct causal order. Despite this, the causal order cannot be necessarily inferred from the timestamps alone.

   In the provided protocol to achieve totally ordered multicast, the effect was that if message "a" causally preceded message "b", then the preceding timestamp would be lower than the following

timestamp for all processes. This is as any process in the causal relationship will increment their logical clock during their hop of the causal chain, when sending a message. This means message "a" will also always be passed to the processes before message "b". Since any event which causally preceded another will always be before the other event in time, where C(a) < C(b), total ordering also achieves a causal ordering.

While -**a**- causal order is achieved with this protocol, precisely which messages are in causal relationships cannot be inferred from the timestamps alone. For instance, in the diagram below, the message from P_2 is not caused by the message from P_0. Just because P_1 received a message from P_0 with a timestamp of 10 and a message from P_2 with timestamp 16, it does not mean the former ***caused*** the latter. From the perspective of P_1, however, since it only has the single timestamps, it has no way of knowing this. There is simply not enough information to tell.



4. Note: the physical communication network need not to be a ring, it may be bus network also. But the processes in a system are logically organized in a ring.

   We will try to order the n processes into the token ring. We will ask all processes to broadcast their processID at the beginning, around the same time. Each indivisual process would also keep another PID, which will be the PID that is immediately after its own. If the process is the one with highest PID, such that there's no PID immediately above its PID, then it will keep the smallest PID from the broadcast (this would be the start of the token ring).

   Time efficiency: we will need to wait for n processes to finish their broadcast, therefore the time efficiency is linear of $O(n)$ in the worst case where not more than one process can broadcast at once. If all the broadcasts can be made and processed completely concurrently, the time efficiency is $O(1)$. In terms of number of messages sent, since every process has to send to every other process, $O(n^2)$ messages must be sent. Space efficiency: each process will store another PID, therefore the space efficiency is still linear of $O(n)$ overall, though each of the "n" process has to commit $O(1)$ memory itself.

5. The alternate method we propose is that instead of the coordinators responding immediately whether a client can or cannot get the resource, they instead all keep a priority queue of all the clients which have asked for the resource. The queue is sorted according to some global unique ID (such as a process ID appended to the timestamp of the client) so there are no collisions and all coordinators will be order them the same way. To enforce total ordering, upon receipt of a client request the coordinators will send acknowledgements to all other coordinators. An element may only be popped off the queue once acknowledgements for it have been received from all other coordinators.

   Whenever that resource is released, the coordinators will pop off the head of the queue, representing the next client it will vote for, and send a positive vote to the client. The client will wait until it receives "m" votes from the coordinators, which will eventually happen due to the priority queues in the coordinators, ensuring progress.

   With this system, all the coordinators that were sent requests by the client will send requests to all "n-1" other coordinators. If we denote the number of coordinators the clients sends a request to as "X", then this is X*(n-1) messages to synchronize the coordinators for voting. The client must wait for "m" positive votes before getting the resource. Assume the network has no losses. If it sends "X" requests out, then

$$msg_{GetResource} = X + X(n-1) + m$$

   In the most optimistic case, "X" will be "m", so the client sends requests to the minimum number of coordinators to get a majority vote. In the worst case, the client must send a request to all "n" coordinators. Recall the "m" is some fraction of "n", where $m = an$

$$min\{msg_{GetResource}\} = m + m(n-1) + m = m(n+1) = an(n+1) \rightarrow \Theta(n^2)$$
$$max\{msg_{GetResource}\} = n + n(n-1) + m = n^2 + m \rightarrow \Theta(n^2)$$

6. A peer will not pass the token unless another peer wants to have it. Whenever a peer wants access to the resource, it will send out a request along the ring (say in a counter clockwise direction). This request will eventually reach the peer who currently has the token. The peer with the token will pass the token counter clockwise around the ring once it has both held onto it for its allowed time slice (if it actually still wants the token) and it has received a request for the token.

   When sending the token, the message will also contain the identifiers for all the peers the peer with the token received requests from. When a peer gets the token message, it can simply forward it if it does not need the resource. If it does want the resource, it will hold onto the token for its time slice, remove itself from the list of peers wanting the resource, and update that list of peers with any requests it receives from other peers.

   This method means that when a peer wants the token, the total number of messages needed for both the request and token is tight bound to exactly "n", $\Theta(n)$, the number of peers in the ring.

7. The output is not legal.

We will define "legal" as not violating program order, with strict consistency. We will also assume that from "001110", "00" belongs to some process A, "11" belongs to some process B, and "10" belongs to some process C. We can imagine the processes as interchangeable as they are identical in that they all set a variable to a value and print out the other two variables. We can imagine the print statements from each of these processes as being (V is variable set by a process):

$$A: print(V_B, V_C)$$

$$B: print(V_A, V_C)$$

$$C: print(V_A, V_B)$$

Since A has "00", its print must have run before the other two processes started, which means it precedes everything. Since B has "11", its print statement must run after both other processes have run their first instructions. Since C has "10", its print statement must be after A runs its first instruction and before B runs its first instruction. We also know that the prints statements for A, B, and C must happen in that order. With this, we can construct the following scheduling graph. The two instructions in each process are separated and numbered. As can be seen, there is a loop involving B1, B2, and C2. This means that there is no way to schedule those three operations without violating the program order. Therefore, this output is not legal.