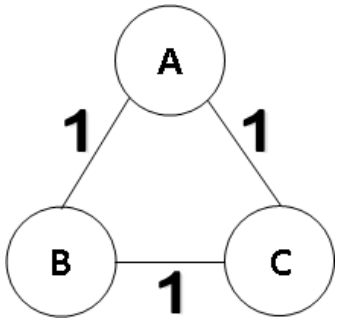
University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

ECE 454
Assignment 3

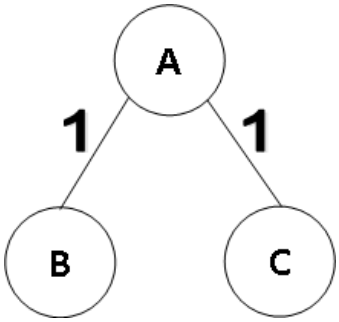
Prepared by
Chan, Carl
UW Student ID Number: 20383063
UW User ID: c73chan@uwaterloo.ca
and
Li, Debin
UW Student ID Number: 20389489
UW User ID: d73li@uwaterloo.ca

10 June 2014

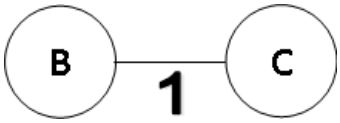
1. This will be disproven by counterexample. Imagine a graph like the one below. The weight of all edges is 1. Suppose we want to create a Steiner tree containing vertices B and C.



The claim is that the smallest sub-tree of an MST of this graph that contains vertices B and C is a Steiner tree for those vertices. To disprove this, create an MST rooted at vertex A like below. As the weight between all three vertices is the same, it is trivial to confirm this is an MST of the graph.



The smallest sub-tree of the above tree containing both vertex B and C is the tree itself, which has a total weight of 2. With access to all the edges in the original graph, however, we can create a Steiner tree for those vertices with a total weight of 1. This can be seen below.



The arbitrary sub-tree found before had a weight greater than the above Steiner tree. This means the MST sub-tree was not a Steiner tree and the claim is not true for an arbitrary MST of an arbitrary graph.

2. This can be proven by induction:

Let "V" be the number of nodes (vertices)

Base Case: V=1. When we only have 1 node, there are no edges. Thus, the claim holds for the base case.

Induction:

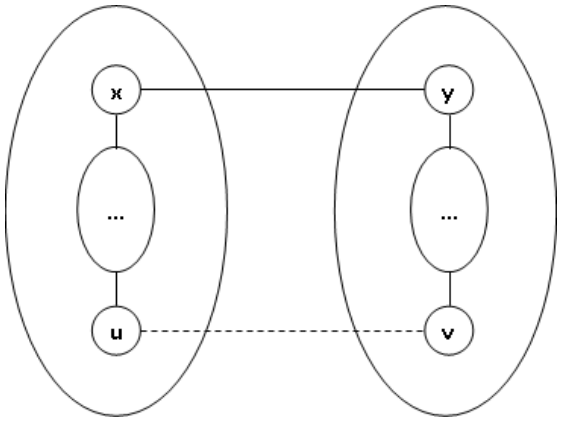
Let V=k, and assume that the statement is true, that there will be k-1 edges.

For V=k+1, there should be k edges.

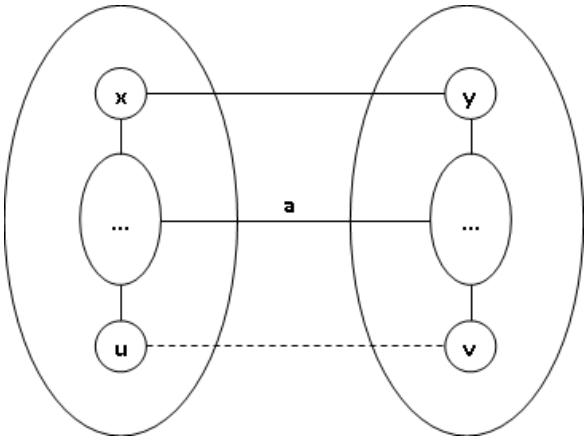
- This is as when a vertex is added to the graph of V=k (which has k-1 edges), it will have to be a leaf node, which will add one edge.
- The addition of the new vertex can only be accompanied by one edge.
- This is due to if more than one edge is added between the new vertex and the original tree (a connected graph) it would stop being a tree due to having a loop.
- The number of edges will thus increase by 1.

Thus, by induction it is shown that E[T]=V[T]-1.

3. Imagine a cut like the one below. We want to remove <x, y> from "T" and add <u, v> as we work toward a new MST of the graph. As "x" and "u" are both on the side of the cut for V\Q (already added vertices), they are connected by the part of the MST "T" on the left cut. In order for <u, v> to be a viable replacement for <x, y>, the resulting graph after replacement must still be a tree. This means that "y" and "v" are connected by the part of "T" on the right side of the cut.

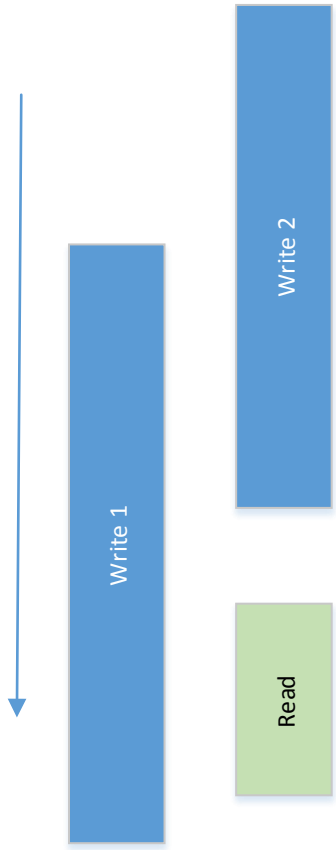


We know that <x, y> is the only such edge in "T" to cross the cut as if we add another edge "a" to "T" which bridges the cut then we end up with a cycle. This is as the edge "a" would in some way be connected to vertices "x" and "y" via the edges in "T" on the respective sides of the cut. As trees are acyclic, this means the MST "T" cannot have more than one edge crossing this cut while still being an MST.



4.

a. As seen in the picture below:



If there is no protection mechanism, any client can request to read or write a file at any time to take effect, even if they overlap in time. This can be a problem as the read and write operations will not have any locking, or be atomic. In the case above, the read operation would violate the UNIX notation of correctness, which is that "a read(file) returns the effect of the last write(file)". In the case above, the read would return the result of "write 2", but with some of the data possibly overwritten by "write 1", so the end result for the read would be something in between the two writes. This violates the correctness condition.

b. The same argument can be made as before. In the above figure, since there are no protection mechanisms the read might pick up changes from parts of both "write 1" and "write 2", even though "write 1" has not finished yet. Thus, we do not satisfy the correctness condition.

c. After getting a start transaction, the server checks if there is a lock on the resources used by that transaction for both reads and writes. If not, that transaction gets the lock and is allowed to work with that resource. If another transaction already has the lock then the younger transaction (with the lower timestamp on the server side) is killed and rolled back so the older one can have the lock. This avoids starvation as the younger transaction can retry after the older transaction has finished, this time becoming an older transaction itself. This means it will eventually be the oldest transaction and can definitely get the locks it needs.

While a more sophisticated system could handle allowing reads to occur while writes are happening, the system described above is sufficient for the UNIX definition of correctness. No

writes can overlap so the integrity of the data is ensured. No reads can overlap with other operations, so since all write transactions are now atomic any read is returning whatever transactions completed before it.

5. One of the clocks are correct, while the other has a certain amount of relative skew from the other:

$$\left|\frac{1000 - 990}{1000}\right| \times 100\% = 1\%\ clock\ skew$$

The maximum clock skew occurs after the maximum amount of time has passed without synchronizing the clocks. In this case, this is 60 seconds as the synchronizations happen every minute:

$$1\% \times \frac{60\ s}{1\ min.} = 0.6\ seconds$$

Therefore, the maximum clock skew is 0.6 seconds.

6. It is not necessary for every message to be acknowledged. That is, there exists a case where correctness and progress are maintained even without acknowledgements. Imagine a case where no acknowledgements at all are used with Lamport clocks. In order to achieve totally ordered multicast, all that is needed is that the messages arrive at each of the processes' queues in the chronologically correct order (for instance if messages reach receivers instantaneously). Timestamps within the message can be used to achieve this. As long as the messages are passed to the applications as soon as they arrive then progress will be ensured as well. Thus, there is an instance where it is possible to get correctness and progress without using acknowledgements.

   That acknowledgements are not strictly necessary, however, does not mean correctness and progress can be *guaranteed* without them. In fact, it is probable that the system will not work without acknowledgements. By the definition of "Totally Ordered Multicast", all messages need to be received and processed in the same order by all receivers. Requiring acknowledgements from all the processes means that a process will not get to process a message until it knows for certain that all the processes have the same message ready as well. This can be important for cases where messages to certain processes are either lost or delayed.

   Consider a message X sent to multiple processes, which is delayed for one of the processes P. If the other processes do not wait for the acknowledgement from P, they might process the message before P gets it. Suppose there is another message Y which is sent after X, but reaches P before X. In this case without acknowledgements, P will process Y before X since it has not received the latter yet, while the other processes will process X before Y. This difference violates totally-ordered multicasting. This would not happen if the processes wait for all the others to acknowledge getting the message before processing it.