Assignment 8

Problem Statement Consider the problem of implementing a timestamp object in an asynchronous message passing system with at most one Byzantine process. Whenever a non-faulty process performs GetTS, it must eventually get a timestamp which is larger than all the timestamps returned by GetTS operations performed by non-faulty processes that finished before this GetTS began.

For what number of processes, n, is it possible to solve this problem? Justify your answer.

Claim 1. *It is possible to implement a timestamp object when $n = 1$ or when $n > 3$.*

Proof. It is trivial to implement a timestamp object for one process. The process keeps a local counter and increments it for every call to GetTS. Lemma 2 shows that it is impossible to implement the timestamp object when $n = 2$ and $n = 3$.

In the following we will present an algorithm which implements a timestamp object in an asychronous message passing system with at most one Byzantine process when there are ≥ 4 processes. Denote the processes by $p_1, ..., p_n$. The general idea of this algorithm will be similar to that of simulating shared registers using message passing systems (Attiya Section 10.4).

A high-level description of the algorithm is as follows: every process p_i has a local variable t_i which stores the largest timestamp p_i has seen and a vector S_i to store proposed timestamps. t_i is initialized to zero and S_i is empty at the start of a GetTS operation. When p_i executes GetTS, p_i broadcasts message request to all other processes (including itself). When a process p_j gets a request message from p_i, it sends p_i message propose(t_j). Upon receiving propose(t), p_i adds t to S_i. When $|S_i| \geq n-1$ process p_i sets $t_i \leftarrow \max\{t \in S_i\} + 1$ and broadcasts message confirm(t_i) to all other processes (again, including itself). Upon receiving message confirm(t_i) from p_i, p_j updates $t_j \leftarrow \max(t_i, t_j)$ and sends back $\langle ACK \rangle$. p_i counts the number of $\langle ACK \rangle$ messages that it has received. When this number hits $n-1$, p_i empties S_i and outputs t_i.

It remains to show that a GetTS operation g_i which finished before the start of another GetTS operation g_j began will return a smaller timestamps than the one returned by g_j. Suppose p_i performed g_i and received t_i and then p_j performed g_j and received t_j (it is possible that $p_i = p_j$ but then the condition is satisfied trivially). Before g_i ended, p_i sent t_i to all processes and received $n-1$ $\langle ACK \rangle$ messages. Thus all processes p_k which successfully returned $\langle ACK \rangle$ must have $t_k \geq t_i$. If p_j sent $\langle ACK \rangle$ to p_i in g_i then $t_j \geq \max\{t \in S_j\} + 1 \geq t_i + 1 > t_i$. Suppose not and consider the contents of S_j just before p_j updates t_j during g_j. $|S_j| \geq n-1$ so p_j must have received $n-2$ messages from other processes. Since $n \geq 4$, p_j must have received a message from a non-faulty process p_k. p_k sent $\langle ACK \rangle$ to p_i during g_i so $t_k >\geq t_i$. Thus $t_j = \max\{t \in S_j\}+1 \geq t_k+1 > t_i$. □

Lemma 2. *When there are 2 or 3 processes and at most one Byzantine process, it is impossible to implement a timestamp object in an asynchronous message passing system.*

Proof. It is easy to see that there cannot be an implementation of a time stamp object when $n = 2$. The adversary simply delays all messages sent between the two processes p_0 and p_1. From the perspective of p_i, it is unclear if p_{1-i} is byzantine or just slow. Thus the value p_i outputs from a GetTS operation is independent of what p_{i-1} outputs.

Next consider the case when $n = 3$. Suppose for a contradiction that there exists an implementation of a timestamp object in the stated model. Let the processes be denoted p_0, p_1, and p_2. Let the initial configuration be C. The adversary will delay all messages passed between p_0 and p_1.

Suppose p_0 performs a GetTS from C. Since p_0 cannot determine if p_1 is Byzantine (and is pretending to crash) or just slow, p_0 must decide upon some timestamp s_0 at time t_0. Let α_0 be the execution where p_0, interacting only with p_2, gets s_0 at t_0. Next suppose that p_1 performs a GetTS operation after t_0. Since messages passed between p_0 and p_1 are still delayed, p_1 also cannot determine if p_0 is Byzantine or just slow so p_1 must decide upon some s_1 at time $t_1 > t_0 + 1$. Let α_1 be the execution where p_1, interacting only with p_2, gets s_1 at t_1. Since the GetTS operation by p_0 finished before the start of the GetTS operation by p_1, $s_0 < s_1$.

Suppose instead that p_1 performs GetTS from C. p_2 will behave just as it did in α_1. Since $(C\alpha_0)\alpha_1 \overset{p_1}{\sim} C\alpha_1$, p_1 must decide upon timestamp s_1 at time t_1'. Next p_0 performs GetTS at some time t_0' after t_1'. p_2 will behave just as it did in α_0. Again, since $C\alpha_0 \overset{p_0}{\sim} (C\alpha_1)\alpha_0$, p_0 must decide upon timestamp s_0. This is a contradiction since the GetTS operation by p_1 finished before the start of the GetTS operation by p_0. $\square$