

# Principles of Distributed Computing

## Exercise 6: Sample Solution

### 1 Linear Arrow

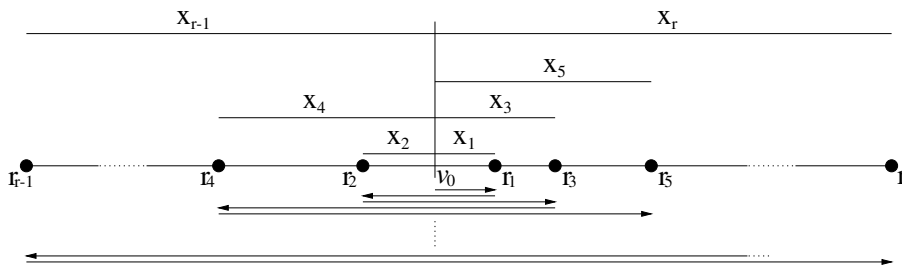


Figure 1: Linear Arrow: Worst Case

As in the lecture, we consider the length of the path on the tree (linear list) when visiting all requests in the order in which they are queued by Arrow. This gives exactly the message complexity of the Arrow protocol. At the beginning, the arrows point to a specific node  $v_0$  (the root of the initial tree). Let the  $i^{\text{th}}$  request in Arrow's order be  $r_i$  and the total number of requests be  $r$ , i.e. Arrow starts at  $v_0$  and then visits  $r_1, r_2, r_3, \dots, r_r$ . In the worst case, Arrow has to change the side with respect to  $v_0$  in every step (see Figure 1 as an example). We want to evaluate the cost of Arrow for this case. Let  $x_i$  be the distance (i.e. number of hops) between  $r_i$  and  $v_0$ . We start at  $v_0$  and go to  $r_1$ . This costs  $x_1$ . From there, we go to  $r_2$  which causes additional cost of  $x_1 + x_2$ . Going from  $r_2$  to  $r_3$  then costs  $x_2 + x_3$ . In general, going from  $x_i$  to  $x_{i+1}$  costs  $x_i + x_{i+1}$  and therefore, the total cost of Arrow is:

$$\text{cost}(\text{Arrow}) = x_1 + (x_1 + x_2) + (x_2 + x_3) + \dots + (x_{r-1} + x_r) = 2 \sum_{i=1}^{r-1} x_i + x_r.$$

The cost of an optimal queueing strategy is at least

$$\text{cost}(\text{Optimal}) \geq x_{r-1} + x_r.$$

We know that Arrow always visits the nearest of the remaining requests. Therefore, we have that  $x_1 \leq x_2, x_1 + x_2 \leq x_3 - x_1$  and in general

$$x_i + x_{i+1} \leq x_{i+2} - x_i. \tag{1}$$

We sum Inequality (1) over all  $i$  from 1 to  $r - 2$  and get

$$\begin{aligned} \sum_{i=1}^{r-2} (x_i + x_{i+1}) &\leq \sum_{i=1}^{r-2} (x_{i+2} - x_i) \\ x_1 + 2 \sum_{i=2}^{r-2} x_i + x_{r-1} &\leq x_{r-1} + x_r - x_1 - x_2. \end{aligned} \tag{2}$$

By adding  $x_1 + x_{r-1} + x_r$  on both sides of (2), we obtain

$$2 \sum_{i=1}^{r-1} x_i + x_r \leq 2(x_{r-1} + x_r) - x_2$$

$$\text{cost}(\text{Arrow}) < 2 \cdot \text{cost}(\text{Optimal}).$$

## 2 Concurrent Arrow

- a) The current version of Algorithm 7.8 does not take into account concurrent READS. When we allow for multiple READ operations at the same time, they might all follow the arrows to a token node (containing the updated version of the shared variable) and thereby traversing some edges multiple times. In the worst case, there might be  $O(n)$  concurrent READS and the competitive ratio degenerates to  $O(n)$ . See Figure 2.

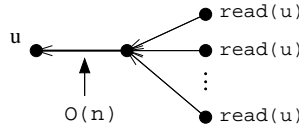


Figure 2: Multiple READ requests in the unmodified protocol can access some edge  $O(n)$  many times.

- b) In order to ameliorate the problem in (a), we can introduce an additional bit at each node that records whether some node has already initiated a READ request for that subtree. If another, concurrent READ encounters a node where that bit is set, it will wait there until the first request comes back and then reads the cached copy as in the original protocol. This way we emulate the sequential access as done in the proof of Theorem 7.9 and obtain a 3-competitive algorithm for concurrent READS (here the WRITES are still sequential and not overlapping with the READS).
- c) Now we additionally allow one WRITE concurrent with the READS. If we simply want an algorithm that *works*, we can reuse the above algorithm and handle WRITES the same way as before (Algorithm 7.8).
- d) The problem we encounter with that approach is sketched in Figure 3. A *linearizable* sequence of concurrent operations has to uphold the following condition: if  $o_1$  terminates before the start of  $o_2$ , then  $o_2$  has to be after  $o_1$  ( $o_1 \rightarrow o_2$ ) in any linearization order.

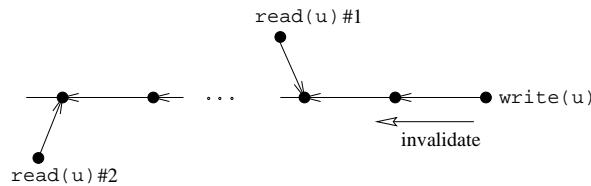


Figure 3: A concurrent WRITE with multiple READS can cause consistency problems.

Now consider a situation as in Figure 3. The first READ<sub>1</sub> operation may well terminate before the second READ<sub>2</sub> operation starts (if the WRITE invalidate takes a long time to travel along the tree), but it will have read the new value  $u_{new}$ . The READ<sub>2</sub> might encounter a cached copy  $u_{old}$  before invalidation. So on one hand we have READ<sub>1</sub>  $\rightarrow$  READ<sub>2</sub> (with perhaps the WRITE in between) due to the linearizability condition, on the other hand we should have READ<sub>2</sub>  $\rightarrow$  WRITE  $\rightarrow$  READ<sub>1</sub> because READ<sub>1</sub> obtains the new value after the WRITE and READ<sub>2</sub> still has the older value before the WRITE. Therefore, our current version of the algorithm is not linearizable.

The solution to the described consistency problem is to split the WRITE into two phases at the expense of one additional message complexity. When a node wants to execute a WRITE,

it first sends an invalidation message to all the cached copies in the tree. Then it waits for a response from all children (essentially an echo on the [cached copies] tree) before it actually writes the new value. This will introduce one more message per edge and the competitive ratio of our concurrent multiple-reads/single-write algorithm is now 4.

- e) When dealing with multiple WRITE requests concurrently, we need to take care to decide which WRITE actually gets to write the value and which ones will simply be ignored. Consider two WRITES meeting in their invalidation phase: one of them wins (somehow) and can continue invalidating the cache tree; the other one will turn back around and echo to the source that it has lost and cannot write its value. The winning WRITE will still have to invalidate the subtree of the losing node since somewhere there might be another WRITE that thinks it has won. In that case, an edge might be invalidated more than once.