Dynamic Analysis of the Arrow Distributed Protocol

Dynamic Analysis of the Arrow Distributed Protocol

Fabian Kuhn, Roger Wattenhofer
{kuhn,wattenhofer}@inf.ethz.ch
Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

Abstract

Arrow is a prominent distributed protocol which globally orders requests initiated by the nodes in a distributed system. In this paper we present a dynamic analysis of the Arrow protocol. We prove that Arrow is $O(\log D)$-competitive, where D is the diameter of the spanning tree on which Arrow operates.

1 Introduction

Ordering events, messages, or processes is at the heart of any distributed system, arising in a multiplicity of applications, such as distributed shared memory, caching, mobile objects, or totally ordered multicast. The Arrow protocol is a simple yet elegant ordering protocol which experimentally outperforms conventional centralized ("home-based") schemes at high concurrency. In this paper, we present the first dynamic analysis of the Arrow protocol. Besides the immediate result ("How efficient is Arrow?") we feel that our paper introduces novel methods to dynamically analyze distributed systems.

In the remainder of the introduction, we familiarize the reader with the Arrow protocol, and compare our results to related work.

1.1 Arrow

We are given a distributed system with n nodes. These nodes are connected through a pre-computed spanning tree T. A node can communicate solely with its direct neighbors in the spanning tree T, by means of message passing. The nodes initiate ordering requests at arbitrary times; an ordering request r can be identified by the tuple $r = (u, t)$ where u is the node that initiates the ordering request, and t is the time when the request is initiated.

For simplifying our analysis we assume our distributed system being synchronous: All the nodes proceed in rounds; in a round, each node might initiate new ordering requests, and each node can exchange a message with each of its neighbors.

The Arrow protocol will globally order all the requests initiated over time and over all the nodes in the system. In particular, each request will eventually find its predecessor request, and each request will eventually be found by its successor request in the total order; the very first request in the total order will learn that it is first. More formally:

Definition 1.1. (Ordering Algorithm) *An ordering algorithm $\mathcal{A}$ is a distributed algorithm which defines a total order on all ordering requests $R \subset V \times \mathbb{N}$ such that in the end each node $v \in V$ knows the predecessors of all requests in $\{v\} \times \mathbb{N}$.*

A total ordering can easily be achieved by a central solution, where each request is routed to a previously established "central" node, which locally orders all the requests and informs the requesting nodes about predecessor and/or successor. Since a central solution does not scale well, distributed algorithms have been proposed – a most promising algorithm is the Arrow protocol.

The Arrow protocol operates as follows: Each node v holds a pointer p_v, which points to v itself or to one of v's neighbors in the spanning tree T. Initially there is a designated root node, such that all pointers in the system point towards the root in the spanning tree T, that is, $p_{\mathrm{root}} = \mathrm{root}$ and every other pointer points to the neighbor closest to the

root. In addition, the root initially stores a dummy token $(\mathrm{root}, 0)$, all other nodes initially store no token. The Arrow protocol guarantees that $p_v = v$ if and only if node v stores a token.

When initiating an ordering request $r = (v, t)$, node v checks whether $p_v = v$. If so, the request r is ordered directly after the request represented by the token stored at v, and the token is replaced with the new token (v, t) (representing the new request). If $p_v \neq v$, node v stores the token (v, t), sends a message "find predecessor for (v, t)" to the neighbor p_v, and redirects p_v to itself ($p_v = v$).

A node u receiving a message "find predecessor for (v, t)" from a neighbor w will do the following: If $p_u = u$ the request (v, t) is ordered directly after the request represented by the token stored at u; node u removes the token and directs its pointer to the sending neighbor, that is $p_u = w$. If on the other hand $p_u \neq u$, node u will simply forward the message "find predecessor for (v, t)" to neighbor p_u and then set $p_u = w$.

Despite its simplicity the Arrow protocol produces a correct total order even in an asynchronous system. In this paper we analyze a synchronous model: If several messages (or requests) arrive at a node at the same time, these messages/requests are processed sequentially in arbitrary order. As discussed in Section 4, our analysis extends to the asynchronous model, making some necessary assumptions.

In this paper we study the two standard cost models in distributed computing: time and message complexity. The latency of a request is the time that elapses from initiation of the request to its completion (i.e. when a token determining the predecessor is found); the time complexity is the sum of latencies of all requests. The message complexity is the total number of messages exchanged; in the synchronous model the message complexity equals the time complexity. Returning the predecessor information to the node that initiated the request costs not more than the "find" process itself, and is therefore omitted in our analysis.

Unfortunately, no online ordering algorithm can be competitive in either time or message complexity against an optimal offline algorithm. For time complexity alone, consider a scenario where nodes initiate requests well-spaced in time. An optimal offline algorithm can always send the predecessor information proactively to the next requesting node, such that each request experiences zero latency, and therefore zero cost.

For message complexity alone, consider a scenario where only two nodes u, v initiate requests. An optimal offline algorithm may order all requests of u before all requests of v, such that a single message is enough to transport the information of the last request of u to the first request of v (who experiences a huge latency, for that matter). Again, no reasonable online algorithm can compete against such a powerful offline algorithm. To escape the problem of not being able to compete with time *or* message complexity, we aim for competing with time *and* message complexity. More formally:

Definition 1.2. (Ordering Cost) *The cost of an ordering algorithm is the sum of the communication complexity (total number of messages) and the total time the nodes have to wait until they know the predecessors after placing a request (total latency).*

Other cost metrics are asymptotically equivalent to the sum of message and time complexity. Apart from other functions of message and time (e.g. max), we could similarly analyze the time complexity (latency) only for the problem of finding the successor instead of the predecessor (or successor *and* predecessor) in the total order.

1.2 Related Work

The Arrow protocol has originally been described in a paper by Raymond [11] and independently by Demmer and Herlihy [1]. The Arrow protocol has been implemented in several systems, most notably in the Aleph Toolkit [4]. In [8] it was experimentally demonstrated that Arrow performs well at high load. In [1] and [10] it was revealed how to implement mobile objects (e.g. shared variables, objects, files) using Arrow, in [7] it was shown how Arrow can be used to implement totally ordered multicast efficiently.

There have been several endeavors analyzing the Arrow protocol. In [1] it was proved that find operations do not backtrack, and therefore the time and message complexity of a find operation is at most D,

the diameter of the spanning tree. In [5] it was shown that Arrow self-stabilizes well in case of faults (e.g. lost messages or state). So far the only analysis allowing concurrent requests was done by Herlihy et al. [6]: In particular a special case is studied where all requests are initiated at time 0. It was demonstrated that most of these concurrent requests will find their predecessor "close-by," more precisely that the total message or time complexity is only a factor $\log |R|$ off the optimal, where R is the set of concurrent requests at time 0. In some sense the analyses in [1] and [6] give results for both extremes of the temporal spectrum, where requests are either well-spaced in time [1] or all at exactly the same time [6].

In this paper we give the first analysis that allows nodes to initiate requests at arbitrary times. We will show that in this most general model the (online) Arrow protocol is at most a $O(\log D)$ factor costlier than an optimal omnipotent offline ordering algorithm, where D is the diameter of the spanning tree. For spanning trees with logarithmic diameter, this translates into a doubly-logarithmic competitive ratio.

Finding a good spanning tree is a thriving research problem that is (with respect to Arrow) complementary to this paper. There is a recent breakthrough result by Emek and Peleg [2] which manages to compute a $O(\log n)$ approximation, meaning that the maximum stretch of the computed spanning tree is at most a logarithmic factor (in the number of nodes) larger than the maximum stretch of an optimal spanning tree (with minimum maximum stretch).

There are a variety of other distributed protocols for ordering. Most notably there is the dynamic distributed manager protocol by Li and Hudak, as implemented in their Ivy system [9]. As Arrow, Ivy uses pointers to give the way to not-yet collected tokens of previous requests. In contrast to Arrow, Ivy needs a complete graph to be operational. A find message will then direct all visited pointers directly towards the requesting node, in order to provide shortcuts for future requests. However, despite this "path shorting" optimization, Ginat et al. [3] proved that in the worst-case the amortized cost of a single request is $\Theta(\log n)$. This is not better than Arrow for a tree with logarithmic diameter [1].

Our analysis features an unforeseen connection with the traveling salesperson problem (TSP); in order to establish our main result we must prove a new approximation result of the TSP nearest-neighbor heuristic [12].

The remainder of the paper is organized as follows. In Section 2 we analyze the Arrow protocol; in particular we show that the online Arrow protocol is only a $O(\log D)$ factor costlier than an optimal omnipotent offline ordering algorithm, D being the diameter of the spanning tree in which the protocol is executed. In Section 3 we show that our analysis is almost tight. We conclude the paper in Section 4.

2 Analysis

Our analysis is organized as follows. First we define the costs of the Arrow protocol and an optimal algorithm. In Subsection 2.2 we give an upper bound on the cost of the Arrow protocol. This is followed by a subsection which bounds the cost of an optimal algorithm with a Manhattan TSP from below. Using a new analysis for the TSP nearest neighbor heuristic in Subsection 2.4 we can finally derive the competitive ratio in Subsection 2.5.

2.1 Cost Measures

Throughout the paper we index the requests $R = \{r_0 = (v_0, t_0), r_1 = (v_1, t_1), \dots\}$ in increasing order with respect to t_i with ties broken arbitrarily, i.e. $i < j \implies t_i < t_j$. (To avoid notational clutter we often refer to the node and time of a request r_i directly as v_i and t_i, respectively.)

We first look at the cost of the Arrow algorithm. Let π_{A} be the order which is induced by Arrow, i.e. $\pi_{\mathrm{A}}(i)$ denotes the index of the i^{th} request in Arrow's order. We introduce $r_0 = (\mathrm{root}, 0)$ representing the "virtual" request (token) at the root; since the initial root token is first in any order, we have $r_{\pi_{\mathrm{A}}(0)} = r_0$.

As proved already in [1] in Arrow each request r_j will find its predecessor r_i on the direct path in the spanning tree. Therefore, the communication cost is given by the distance in the spanning tree $d_{\mathrm{T}}(v_i, v_j)$. As argued in the introduction, the same cost holds for the latency. The cost Arrow brings about for placing

r_j after r_i therefore is

$$c_{\mathrm{A}}(r_i, r_j) := 2d_{\mathrm{T}}(r_i, r_j). \quad (1)$$

For the ordering cost of the Arrow algorithm according to Definition 1.2, we get

$$\begin{aligned} \mathrm{cost}_{Arrow} &= \sum_{i=1}^{|R|} c_{\mathrm{A}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i-1)}) \\ &= 2 \cdot \sum_{i=1}^{|R|} d_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}A(i-1)}). \end{aligned}$$

We now look at the cost of an optimal offline ordering algorithm Opt that has complete knowledge about all the requests R. Clearly, an optimal offline algorithm might order the requests differently: Let π_{O} be the order of Opt. Because also in an optimal algorithm Opt a message has to be sent between the nodes of each subsequent pair of requests, the communication cost is bounded from below by $\sum_{i=1}^{|R|} d_{\mathrm{T}}(r_{\pi_{\mathrm{O}}(i)}, r_{\pi_{\mathrm{O}}(i-1)})$. For latency, we have to take into account the additional knowledge of Opt: For a request $r_i = (v_i, t_i)$ the algorithm already knows the succeeding request $r_j = (v_j, t_j)$, thus at time t_i the algorithm can immediately send a message from v_i to v_j. The message reaches v_j at time $t_i + d_{\mathrm{T}}(v_i, v_j)$ and therefore, the latency at v_j is only $\max\{0, t_i + d_{\mathrm{T}}(v_i, v_j) - t_j\}$. We therefore define the cost $c_{\mathrm{O}}(r_i, r_j)$ (see Figure 1) of ordering r_j after r_i in the π_{O} order as

$$c_{\mathrm{O}}(r_i, r_j) := d_{\mathrm{T}}(v_i, v_j) + \max\{0, t_i - t_j + d_{\mathrm{T}}(v_i, v_j)\} \quad (2)$$

The ordering cost (Definition 1.2) of an optimal algorithm then becomes

$$\mathrm{cost}_{Opt} = \min_{\pi} \left\{ \sum_{i=1}^{|R|} c_{\mathrm{O}}(r_{\pi(i)}, r_{\pi(i-1)}) \right\}. \quad (3)$$

Note that π_{O} is an order which minimizes the sum of Equation (3).

The competitive ratio ρ achieved by the Arrow algorithm is the ratio between the cost of Arrow and the cost of an optimal offline ordering strategy:

$$\rho := \frac{\mathrm{cost}_{Arrow}}{\mathrm{cost}_{Opt}}. \quad (4)$$

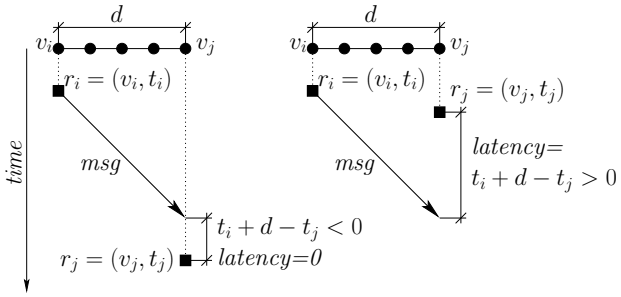

Figure 1: The latency cost of an optimal algorithm for ordering r_j after r_i. Left: v_j is informed about r_i before time t_j, therefore the latency cost is zero. Right: v_j has to wait $t_i + d - t_j$ time units where $d = d_{\mathrm{T}}(v_i, v_j)$.

2.2 The Arrow Protocol in the Dynamic Setting

In this subsection, we have a closer look at the order π_{A} produced by the Arrow algorithm. We will define an additional cost metric c_{T} between requests, for which π_{A} corresponds to a nearest neighbor TSP on the set of requests R. Using amortized analysis we will then show that this new metric c_{T} is comparable to the Arrow cost metric c_{A}.

Definition 2.1. *Let r_i, r_j be two requests such that Arrow orders r_i before r_j, i.e. $\pi_{\mathrm{A}}(r_i) \leq \pi_{\mathrm{A}}(r_j)$. Then the cost metric $c_{\mathrm{T}}(r_i, r_j)$ is defined as*

$$c_{\mathrm{T}}(r_i, r_j) := t_j - t_i + d_{\mathrm{T}}(v_i, v_j).$$

If $\pi_{\mathrm{A}}(r_i) > \pi_{\mathrm{A}}(r_j)$ then $c_{\mathrm{T}}(r_i, r_j) := c_{\mathrm{T}}(r_j, r_i)$.

Lemma 2.2. *The order of Arrow π_{A} is defined by a nearest neighbor TSP tour on the metric c_{T}, starting with the dummy token/request $r_0 = (\mathrm{root}, 0)$ that is initially stored at the* root*. Further, $c_{\mathrm{T}}(r_i, r_j) \geq 0$ for all pairs of requests r_i and r_j.*

Proof. Remember that the spanning tree is initialized with a dummy token $r_0 = (\mathrm{root}, 0)$ stored at the root. Let S be the set of requests which minimize $c_{\mathrm{T}}(r_0, s)$, for $s \in S$. When initiated, the find messages of the requests of S start moving towards the root (at each time, all already initiated requests in S have the same distance from the root). If two or more find requests of S meet at a node, one continues towards the root, and the others are deflected (and dropped from the set S). Thus at least one find

request of S arrives at the root. (If more than one find arrives at the root, all but one are deflected.) By definition this is request $r_{\pi_{\mathrm{A}}(1)}$; it arrives at the root at time $t_{\pi_{\mathrm{A}}(1)} + d_{\mathrm{T}}(v_{\pi_{\mathrm{A}}(1)}, \mathrm{root}) = c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(1)}, r_0)$.

Since $t_{\pi_{\mathrm{A}}(1)} \geq 0$ and $d_{\mathrm{T}}(v_{\pi_{\mathrm{A}}(1)}, \mathrm{root}) \geq 0$ we have $c_{\mathrm{T}}(r_0, r_{\pi_{\mathrm{A}}(1)}) \geq 0$.

Since request $r_{\pi_{\mathrm{A}}(1)}$ minimized $c_{\mathrm{T}}(r_0, r_{\pi_{\mathrm{A}}(1)})$ for all remaining requests $r' \in R - \{r_0, r_{\pi_{\mathrm{A}}(1)}\}$ we have

$$c_{\mathrm{T}}(r_0, r') \geq c_{\mathrm{T}}(r_0, r_{\pi_{\mathrm{A}}(1)}). \tag{5}$$

We need the following helper lemma:

Lemma 2.3. *Consider a starting configuration where the* root *is node $v_{\pi_{\mathrm{A}}(1)}$, and where there was no request $r_{\pi_{\mathrm{A}}(1)}$; call this configuration F_1 and the original configuration F_0. Then no request but $r_{\pi_{\mathrm{A}}(1)}$ will be able to distinguish between configurations F_0 and F_1 during the execution.*

Proof. Configurations F_0 and F_1 are equivalent, except that all pointers on the path P between the original root and $v_{\pi_{\mathrm{A}}(1)}$ are pointing towards the root in F_0 and towards $v_{\pi_{\mathrm{A}}(1)}$ in F_1. Since the request $r_{\pi_{\mathrm{A}}(1)}$ arrived first at every node on the path P in configuration F_0, it has turned all the pointers towards $v_{\pi_{\mathrm{A}}(1)}$ before another request can see the difference between configurations F_0 and F_1. □

For the request $r_{\pi_{\mathrm{A}}(2)}$ we can therefore argue as if we were in configuration F_1, and apply the same techniques as we did for request $r_{\pi_{\mathrm{A}}(1)}$.

To prove $c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(1)}, r_{\pi_{\mathrm{A}}(2)}) \geq 0$ we use inequality (5). (For the sake of generality of the argument we write $r_{\pi_{\mathrm{A}}(0)}$ instead of r_0, $v_{\pi_{\mathrm{A}}(0)}$ instead of root, and $t_{\pi_{\mathrm{A}}(0)}$ instead of 0.)

$$\begin{aligned}
0 &\leq c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(0)}, r_{\pi_{\mathrm{A}}(2)}) - c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(0)}, r_{\pi_{\mathrm{A}}(1)}) \\
&= t_{\pi_{\mathrm{A}}(2)} - t_{\pi_{\mathrm{A}}(0)} + d_{\mathrm{T}}(v_{\pi_{\mathrm{A}}(0)}, v_{\pi_{\mathrm{A}}(2)}) \\
&\quad - \big(t_{\pi_{\mathrm{A}}(1)} - t_{\pi_{\mathrm{A}}(0)} + d_{\mathrm{T}}(v_{\pi_{\mathrm{A}}(0)}, v_{\pi_{\mathrm{A}}(1)})\big) \\
&\leq t_{\pi_{\mathrm{A}}(2)} - t_{\pi_{\mathrm{A}}(1)} + d_{\mathrm{T}}(v_{\pi_{\mathrm{A}}(1)}, v_{\pi_{\mathrm{A}}(2)}) \\
&= c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(1)}, r_{\pi_{\mathrm{A}}(2)})
\end{aligned}$$

(The last inequality is due to the triangle inequality $d_{\mathrm{T}}(v_i, v_j) + d_{\mathrm{T}}(v_j, v_k) \geq d_{\mathrm{T}}(v_i, v_k)$.) For all requests $r_{\pi_{\mathrm{A}}(i)}$ with $i > 1$ we inductively do the same steps, and the lemma follows. □

Lemma 2.4. *Let $r_i = (v_i, t_i) \in R$ and $r_j = (v_j, t_j) \in R$ be two requests. If $t_j - t_i > d_{\mathrm{T}}(v_i, v_j)$, r_i is ordered before r_j by Arrow.*

Proof. Follows from the $c_{\mathrm{T}}(r_i, r_j) \geq 0$ discussion in Lemma 2.2. □

Lemma 2.5. *Let C_T be the cost of ordering all requests in Arrow's order π_{A} with respect to c_{T}. We have*

$$\mathrm{C}_T = \frac{1}{2}\mathrm{cost}_{Arrow} + t_{\pi_{\mathrm{A}}(|R|)},$$

where $t_{\pi_{\mathrm{A}}(|R|)}$ is the request time of the last request in π_{A}.

Proof. We show the lemma by induction over the requests in π_{A}. Let $c_A(i)$ and $c_T(i)$ denote the costs of cost_{Arrow} and C_T up to the i^{th} request. We show that we have

$$c_T(i) = \frac{1}{2}c_A(i) + t_{\pi_{\mathrm{A}}(i)} \tag{6}$$

for all $i \in \{0, |R|\}$. Request r_0 is again the "virtual" request at time 0, and therefore, Equation (6) is clear for $i = 0$. For the induction step, we have a look at $c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i+1)})$ for two succeeding requests. We have

$$c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i+1)}) = t_{\pi_{\mathrm{A}}(i+1)} - t_{\pi_{\mathrm{A}}(i)} + \frac{c_{\mathrm{A}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i+1)})}{2}$$

and therefore

$$\begin{aligned}
c_T(i+1) &:= c_T(i) + c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i+1)}) \\
&= \frac{c_A(i)}{2} + t_{\pi_{\mathrm{A}}(i)} + c_{\mathrm{T}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i+1)}) \\
&= \frac{c_A(i) + c_{\mathrm{A}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i+1)})}{2} + t_{\pi_{\mathrm{A}}(i+1)}
\end{aligned}$$

which completes the proof. □

Assume we are given a set of requests where times of high activity alternate with times where no request is placed. Intuitively, it seems apparent that the foremost ordering differences between Arrow and an optimal offline algorithm are in the high activity regions. Neglecting the order inside high activity regions, Arrow and the offline algorithm essentially produce the same ordering. In Lemma 2.6, we show

that if after some request r no request occurs for a long enough time, we can shift all requests occurring after r back in time without changing the cost of both Arrow and the offline algorithm.

Lemma 2.6. *Let $r_i = (v_i, t_i)$ and $r_{i+1} = (v_{i+1}, t_{i+1})$ be two consecutive requests w.r.t. time of occurrence. Further choose two requests $r_a \in R_{\leq t_i}$ and $r_b \in R_{\geq t_{i+1}}$ minimizing $\delta := t_b - t_a - d_{\mathrm{T}}(v_a, v_b)$. If $\delta > 0$, all requests $r = (v, t) \in R_{\geq t_{i+1}}$ can be replaced by $r' := (v, t - \delta)$ without changing the cost of Arrow and without increasing the cost of an optimal offline algorithm.*

Proof. By Lemma 2.4, the nodes in $R_{\leq t_i}$ are ordered before the nodes in $R_{\geq t_{i+1}}$ by Arrow. By the definition of δ this does not change. The transformation therefore does not change the ordering of the the nodes in $R_{\leq t_i}$. Let r be the last request of $R_{\leq t_i}$ in Arrow's order. All costs $c_{\mathrm{T}}(r, r')$ between r and requests $r' \in R_{\geq t_{i+1}}$ are decreased by δ. Therefore, request r'_0 minimizing $c_{\mathrm{T}}(r, r')$ among all $r' \in R_{\geq t_{i+1}}$ remains the same. Clearly, the order of the nodes in $R_{\geq t_{i+1}}$ is not changed as well and thus, Arrow's order remains unchanged under the transformation of the lemma. Because the cost c_{A} of Arrow only depends on the order (see (1)), c_{A} remains unchanged under the transformation.

For the optimal offline algorithm, we show that the cost $c_{\mathrm{O}}(r, r')$ between any two consecutive requests $r = (v, t)$ and $r' = (v', t')$ cannot be increased by the transformation. If both r and r' are either in $R_{\leq t_i}$ or in $R_{\geq t_{i+1}}$, $c_{\mathrm{O}}(r, r')$ does not change. If $r \in R_{\geq t_{i+1}}$ and $r' \in R_{\leq t_i}$, the edge goes back in time and therefore $c_{\mathrm{O}}(r, r')$ is reduced by δ. If $r \in R_{\leq t_i}$ and $r' \in R_{\geq t_{i+1}}$, by the definition of δ, the latency remains zero and therefore $c_{\mathrm{O}}(r, r')$ does not change. □

In the following we assume that all requests are already transformed according to Lemma 2.6.

Lemma 2.7. *Let $r_i = (v_i, t_i)$ and $r_{i+1} = (v_{i+1}, t_{i+1})$ be two consecutive requests w.r.t. time of occurrence. Without loss of generality, we can assume that there are requests $r_a \in R_{\leq t_i}$ and $r_b \in R_{\geq t_{i+1}}$ for which $d_{\mathrm{T}}(r_a, r_b) \geq t_b - t_a$*

Proof. If it is not the case, we apply the transformation of Lemma 2.6 as many times as necessary. □

Lemma 2.8. *The cost $c_{\mathrm{T}}(r_i, r_j)$ of the longest edge (r_i, r_j) on Arrow's tour is $c_{\mathrm{T}}(r_i, r_j) \leq 3D$ where D is the diameter of the spanning tree T.*

Proof. For the sake of contradiction, assume there is an edge (r_i, r_j) with cost $c_{\mathrm{T}}(r_i, r_j) > 3D$ on Arrow's tour. By Lemma 2.7, we can assume that the temporal difference between two successive requests (w.r.t. time of occurrence) is at most D. Consequently, in each time window of length D, there is at least one request. We set $\varepsilon := (c_{\mathrm{T}}(r_i, r_j) - 3D)/2$. There is a request r_k with $t_k \in [t_i + D + \varepsilon, t_i + 2D + \varepsilon]$. We have

$$t_k - t_i = D + \varepsilon > d_{\mathrm{T}}(v_i, v_k)$$

and therefore by Lemma 2.4, Arrow orders r_i before r_k. Consequently, if $c_{\mathrm{T}}(r_i, r_k) < c_{\mathrm{T}}(r_i, r_j)$, r_j cannot be the successor of r_i and thus (r_i, r_j) cannot be an edge of the Arrow tour. We have

$$\begin{aligned} c_{\mathrm{T}}(r_i, r_k) &= t_k - t_i + d_{\mathrm{T}}(v_i, v_k) \\ &\leq 2D + \varepsilon + D = c_{\mathrm{T}}(r_i, r_j) - \varepsilon. \end{aligned}$$

□

2.3 Optimal Offline Ordering and the Manhattan Metric TSP

In this subsection, we show that (up to a constant factor) the real cost (using c_{O}) of an optimal offline algorithm is the same as the Manhattan cost c_{M} for the same ordering.

Definition 2.9. (Manhattan Metric) *The Manhattan metric $c_{\mathrm{M}}(r_i, r_j)$ is defined as*

$$c_{\mathrm{M}}(r_i, r_j) := d_{\mathrm{T}}(v_i, v_j) + |t_i - t_j|.$$

Lemma 2.10. *Let π be an ordering and C_O and C_M be the costs for ordering all requests in order π with respect to c_{O} and c_{M}. The Manhattan cost is bounded by*

$$C_M \leq 2C_O + t_{\pi(|R|)}.$$

Proof. We can lower bound the optimal cost of (2) by

$$c_{\mathrm{O}}(r_i, r_j) \geq d_{\mathrm{T}}(r_i, r_j) + \max\{0, t_i - t_j\}. \quad (7)$$

Let $D_T = \sum_{i=1}^{|R|} d_{\mathrm{T}}(v_{\pi(i-1)}, v_{\pi(i)})$. Summing up all the $\max\{0, t_i - t_j\}$ using an amortized argument yields

$$\begin{aligned} 2C_O &\geq D_T + 2\sum_{i=1}^{|R|} \max\{0, t_{\pi(i-1)} - t_{\pi(i)}\} \\ &= D_T + \sum_{i=1}^{|R|} |t_{\pi(i)} - t_{\pi(i-1)}| - t_{\pi(|R|)} \\ &\geq C_M - t_{\pi(|R|)}. \end{aligned}$$

(The first and the last inequality follow from the definitions of C_O and C_M, respectively.) □

Lemma 2.11. *Let π be an order C_M be the Manhattan cost for ordering all requests in order π. We have*

$$C_M \geq \frac{3}{2} t_{|R|}$$

where $t_{|R|}$ is the largest time of any request in R.

Proof. Let p be the path connecting the requests R in order π. We define $\alpha(t)$ to be the number of edges of p crossing time t, i.e.

$$\alpha(t) := \left|\{(r_{\pi(i)}, r_{\pi(i+1)}) \in p \,|\, t \in [t_{\pi(i)}, t_{\pi(i+1)}]\}\right|.$$

Further, $\alpha(t', t'')$ denotes the maximum $\alpha(t)$ for any $t \in [t', t'']$. We partition R into subsets $R_1, \ldots, R_k$ where the R_i are maximal subsets of consecutive (w.r.t. time of occurrence) requests for which $\alpha(t) \geq 2$.

Let $R_i := \{r_{i,1}, \ldots, r_{i,s_i}\}$ where the $r_{i,j}$ are ordered according to $t_{i,j}$, i.e. $j' > j \rightarrow t_{i,j'} > t_{i,j}$. We have $r_{1,1} := r_0$, $r_{i+1,1}$ is the request occurring next after r_{i,s_i}, and r_{i,s_i} is the latest request in $R_{>t_{i,1}}$ for which $\alpha(t_{i,1}, t_{i,s_i}) \geq 2$. If there is no request in $R_{>t_{i,1}}$ for which $\alpha(t_{i,1}, t_{i,s_i}) \geq 2$, $r_{i,s_i} := r_{i,1}$.

The Manhattan cost $c_{\mathrm{M}}(r_a, r_b)$ consists of two separate parts, the distance cost $d_{\mathrm{T}}(v_a, v_b)$ and the time cost $|t_b - t_a|$. Let c_{M_d} and c_{M_t} denote the total distance and time costs of c_{M}, respectively, i.e. $c_{\mathrm{M}} = c_{M_d} + c_{M_t}$. To get a bound on c_{M_t}, we define $\Delta t_i^{(1)}$ and $\Delta t_i^{(2)}$ as follows:

$$\Delta t_i^{(2)} := t_{i,s_i} - t_{i,1} \text{ and } \Delta t_i^{(1)} := t_{i+1,1} - t_{i,s_i}.$$

By the definition of the R_i, we have

$$c_{M_t} \geq 2\sum_{i=1}^{k} \Delta t_i^{(2)} + \sum_{i=1}^{k-1} \Delta t_i^{(1)}. \tag{8}$$

We now show how to get a lower bound on c_{M_d}. First, we observe that path p consists of the edges connecting requests inside the R_i as well as one edge per pair R_i and R_{i+1} connecting a request in R_i with a request in R_{i+1}. Thus, path p first visits all nodes of R_1, then all nodes of R_2, and so on.

Let r_a and r_b be two requests for which $t_b - t_a \leq d_{\mathrm{T}}(v_a, v_b)$. Assume that $r_a \in R_i$ and $r_b \in R_j$ for $j > i$. Further let $c_{M_d}(i, j)$ be the total distance cost occurring between requests of $R_i \cup \cdots \cup R_j$. Because r_a and r_b have to be connected by p we have

$$c_{M_d}(i, j) \geq d_{\mathrm{T}}(v_a, v_b) \geq \sum_{\ell=i}^{j-1} \Delta t_\ell^{(1)}. \tag{9}$$

By Lemma 2.7, we can assume for each i there are requests $r_a \in R_{\leq t_{i,s_i}}$ and $r_b \in R_{\geq t_{i+1,1}}$ for which $t_b - t_a \leq d_{\mathrm{T}}(v_a, v_b)$. We can choose r_a's and r_b's such that all $\Delta t_i^{(1)}$s are covered and such that each R_i is covered at most twice. We start by choosing $r_{a,1}$ and $r_{b,1}$ such that $r_{a,1} \in R_1$ and such that $t_{b,1}$ is as large as possible. Assume that $r_{b,i-1}$ is in R_j. $r_{a,i}$ and $r_{b,i}$ are chosen such that $t_{a,i} \leq t_{j,s_j}$ and such that $t_{b,i}$ is as large as possible. We stop as soon as $r_{b,i} \in R_k$. By Lemma 2.7, we make progress in each step and therefore, the last t_b will be in R_k.

Let R_j be the subset containing $t_{b,i}$. By the way we choose the $t_{a,i}$ and $t_{b,i}$, it is guaranteed $r_{a,i+2}$ is in a subset $R_{j'}$ for which $j' > j$. If this were not the case, $r_{a,i+2}$ and $r_{b,i+2}$ would have been chosen instead of $r_{a,i+1}$ and $r_{b,i+1}$. If we sum up the estimates of Equation (9) for all pairs $r_{a,i}$ and $r_{b,i}$, each edge is at most counted twice and therefore

$$c_{M_d} \geq \frac{1}{2}\sum_{i}^{k-1} \Delta t_i^{(1)}.$$

Combining this with Equation (8) concludes the proof. □

Lemma 2.12. *Let π be an order and C_O and C_M be the costs for ordering all requests in order π with respect to c_{O} and c_{M}. The Manhattan cost is bounded by*

$$C_M \leq 6C_O.$$

Proof. By the Lemmas 2.10 and 2.11, we have

$$\frac{3}{2} t_{|R|} \leq 2C_0 + t_{|R|}$$

and therefore $t_{|R|} \leq 4C_O$. (Note that $t_{|R|} \geq t_{\pi(|R|)}$.) Applying this to Lemma 2.10 completes the proof. □

2.4 The TSP Nearest Neighbor Heuristic

We have seen that the cost of the Arrow protocol is closely related to the nearest neighbor heuristic for the TSP problem. In [12], it has been shown that a nearest neighbor tour is only by a factor $\log N$ longer than an optimal tour on a graph with N nodes. We cannot use this result for two reasons. First, the number of requests $|R|$ (the nodes of the tour) is not bounded by any property of the tree T (e.g. number of nodes n, diameter D). The number of requests may grow to infinity even if there are no two requests which are handled concurrently by Arrow. Second, and more important, the nearest neighbor tour of Arrow is with respect to the cost c_{T} for which the triangle inequality does not hold. However, this is a necessary condition for the analysis of [12]. Here, we give a stronger and more general approximation ratio for the nearest neighbor heuristic, removing both described problems.

Theorem 2.13. *Let V be a set of $N := |V|$ nodes and let $d_n : V \times V \to \mathbb{R}$ and $d_o : V \times V \to \mathbb{R}$ be distance functions between nodes of V. For d_n and d_o, the following conditions hold:*

$$d_o(u,v) = d_o(v,u), \;\; d_n(u,v) = d_n(v,u)$$
$$d_o(u,v) \geq d_n(u,v) \geq 0, \;\; d_o(u,u) = 0$$
$$d_o(u,w) \leq d_o(u,v) + d_o(v,w)$$

Let C_N be the length of a nearest neighbor TSP tour with respect to the distance function d_n and let C_O be the length of an optimal TSP tour with respect to the distance function d_o. Then

$$C_N \leq \frac{3}{2}\lceil \log_2(D_{\mathrm{NN}}/d_{\mathrm{NN}}) \rceil \cdot C_O$$

holds, where D_{NN} and d_{NN} are the lengths of the longest and the shortest non-zero edge on the nearest neighbor tour with respect to d_n.

Proof. According to their lengths, we partition the edges of non-zero length of the nearest neighbor (NN) tour in $\log_2(D_{\mathrm{NN}}/d_{\mathrm{NN}})$ classes. Class $\mathcal{C}_i$ contains all edges (u,v) of length $2^{i-1}d_{\mathrm{NN}} \leq d_n(u,v) < 2^i d_{\mathrm{NN}}$, i.e. the lengths of all edges of a certain class differ by at most a factor 2. We show that for each class the sum of the lengths of the edges is at most $3/2 \cdot C_O$. We therefore look at a single class $\mathcal{C}$ of edges. Let d be the length of the shortest edge (w.r.t. d_n) of $\mathcal{C}$. All other edges have at most length $2d$.

Let $V_{\mathcal{C}}$ be the set of nodes from which the NN tour traverses the edges of $\mathcal{C}$. We compare the total length of the edges in $\mathcal{C}$ to the length (w.r.t. d_o) of an optimal TSP tour t on the nodes of $V_{\mathcal{C}}$. Because of the triangle inequality the length of such a tour is smaller than or equal to C_O. Consider an edge (u,v) of the tour t. W.l.o.g., assume that in the NN order, u comes before v. Let u' be the successor of u on the NN tour. The edge (u,u') is in $\mathcal{C}$. During the NN algorithm, at node u, v could have been chosen too. Therefore, $d_n(u,u') \leq d_n(u,v) \leq d_o(u,v)$. Thus, for every edge e on the optimal tour t, there is an edge e' on the NN tour whose length is smaller than or equal to the length of e. Because e and e' have one end-point in common, the length of tour t is at least twice the sum of the lengths of the $\lceil |\mathcal{C}|/2 \rceil$ smallest edges of $\mathcal{C}$. Because the length of all edges in $\mathcal{C}$ is at most $2d$, the sum of the lengths of all edges in $\mathcal{C}$ is at most 3 times the sum of the edges of the small half. This completes the proof. □

2.5 Complexity of Arrow

In this final subsection we put our individual parts together .

Theorem 2.14. *Let cost_{Arrow} be the total cost of the Arrow protocol and let cost_{Opt} be the total cost of an optimal offline ordering algorithm. We have*

$$\rho = \frac{\mathrm{cost}_{Arrow}}{\mathrm{cost}_{Opt}} = \mathrm{O}(\log D)\,,$$

where D is the diameter of the spanning tree T.

Proof. We show that

$$C_T \leq \left(\frac{3}{2}\lceil \log_2(3D) \rceil + 1\right) C_M. \qquad (10)$$

The lemma then follows by the Lemmas 2.5 and 2.12. Equation (10) can be derived from the TSP

nearest neighbor result of Theorem 2.13 as follows. c_{T} and c_{M} comply with the conditions for $d_n(u,v)$ and $d_o(u,v)$, respectively. By Lemma 2.2, $c_{\mathrm{T}} \geq 0$. Further, by the definition of c_{T}, we have

$$\begin{aligned} c_{\mathrm{T}}(r_i, r_j) &= t_j - t_i + d_{\mathrm{T}}(v_i, v_j) \\ &\leq |t_j - t_i| + d_{\mathrm{T}}(v_i, v_j) = c_{\mathrm{M}}(r_i, r_j). \end{aligned}$$

Clearly, the triangle inequality holds for the Manhattan metric c_{M}. The only thing missing to apply Theorem 2.13 is a bound on the ratio of the longest and the shortest edge on Arrow's NN path. By Lemma 2.8, the maximum cost of any edge on Arrow's path is $3D$. The minimum non-zero cost of an edge is 1 because time is an integer value (we have a synchronous system). Lemma 2.13 is about TSP tours (i.e. connecting request $r_{\pi(|R|)}$ again with r_0). Since the last edge of a tour has at most the cost of the whole path, there is an additional factor 2. □

3 Lower Bound

In this section we prove that our analysis is almost tight for any spanning tree.

Theorem 3.1. *For any spanning tree T there is a set of ordering requests R such that the cost of the Arrow protocol is a factor $\Omega(\log D / \log\log D)$ off the cost of an optimal ordering, D being the diameter of the spanning tree T.*

Proof. It is sufficient to concentrate on the nodes on the path P that embody the diameter D of the spanning tree T. Let $v_0, v_1, \ldots, v_D$ be nodes of path P. We recursively construct a set of ordering requests by the nodes of P; nodes outside P do not initiate any ordering requests. For simplicity assume that the initial root is node v_0 (if not, let node v_0 initiate an ordering request well before the other nodes); for simplicity further assume that D is a power of two (if not, drop the part of P outside the largest possible power of two).

Let k be an even integer we specify later. We start the recursion with an ordering request r by node v_D at time k. Request r is of "size" $\log D$ and "direction" $(+1)$; we write $r = (v_D, k, \log D, +1)$ in short. In general a request $r = (v_i, t, s, d)$ with $t > 0$ asks for s requests of the form

$$(v_{i-d\cdot 2^j}, t-1, j, -d), \text{ for } j = 0, \ldots, s-1.$$

In addition to these recursively defined requests there will be requests at nodes v_0 and v_D at times all $0, 1, \ldots, k-1$ (some of these requests are already covered by the recursion). An example is given in Figure 2.

For this set of requests, from the definition of the recursion and as shown in Figure 2, Arrow will order the requests according to their time, i.e. a request with time t_i will be ordered earlier than a request with time t_j if $t_i < t_j$. Requests with the same time t are ordered "left to right" if t is even, and "right to left" if t is odd. Then the cost of Arrow is $\mathrm{cost}_{Arrow} = 2kD$.

The Minimum Spanning Tree (MST) of the requests with the Manhattan Metric is given by a "comb"-shaped tree: Connect all requests at time 0 by a "horizontal" chain, and then connect all requests on the same node (but different request times) by a "vertical" chain, for each node. The Manhattan cost of the MST is D for the horizontal chain. A vertical chain of node v_i costs as much as the latest request of node v_i.

From the recursion we know that there is one request at time k of size $\log D$. Since the recursion only generates requests of smaller size, we have $\log D$ requests at time $k-1$, less than $\log^2 D$ requests at time $k-2$, etc. The Manhattan cost of the MST is therefore bounded from above by

$$\begin{aligned} C_M(MST) &\leq D + \sum_{t=0}^{k} (t \cdot \log^{k-t} D) \\ &< D + \frac{\log^{k+1} D}{(\log D - 1)^2}. \end{aligned}$$

Setting $k = \lceil \log D / \log\log D \rceil$ we get $C_M(MST) = O(D)$ for a sufficiently large D. Since an MST approximates an optimal order π_{O} within a factor of two, and using the fact that cost_{Opt} is up to constants bounded from above by the Manhattan cost (see (2) and Definition 2.9), we conclude that $\mathrm{cost}_{Opt} = O(D)$. Then the competitive ratio is

$$\rho = \frac{\mathrm{cost}_{Arrow}}{\mathrm{cost}_{Opt}} = \frac{2kD}{O(D)} = \Omega(k).$$

□

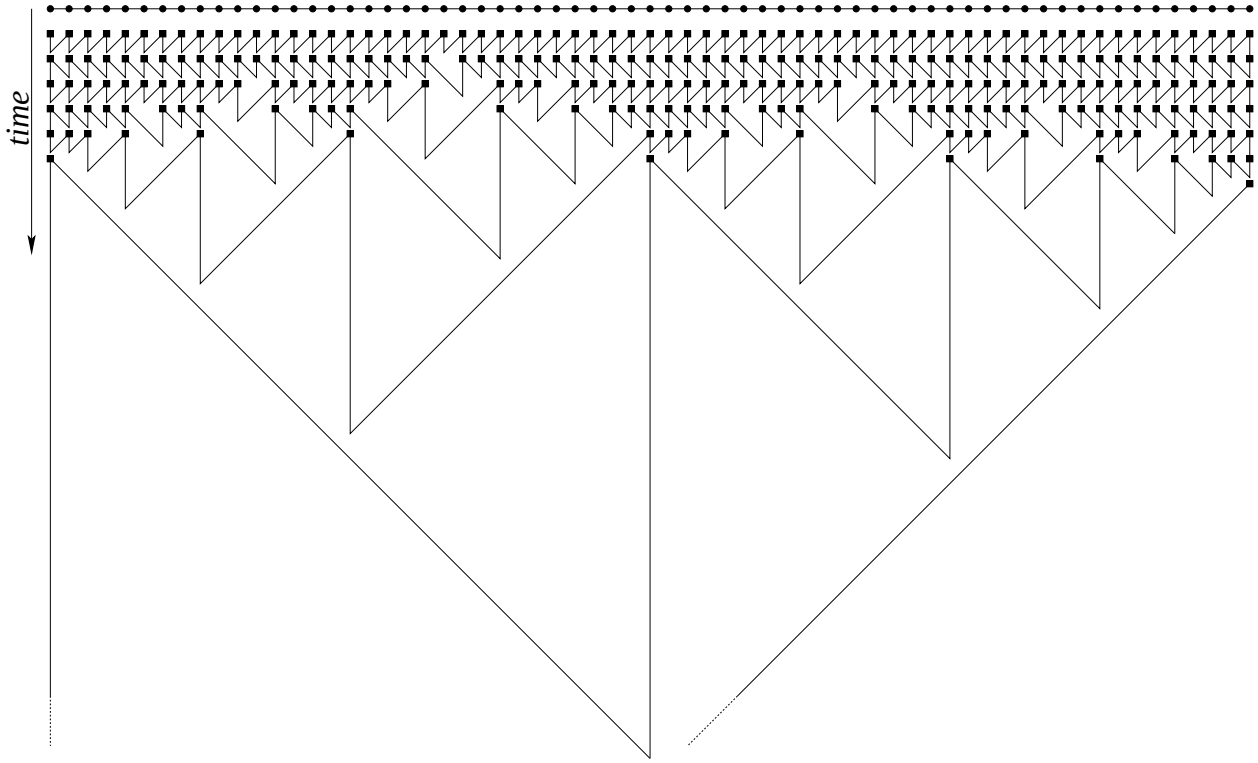

Figure 2: A problem instance with diameter $D = 64$ and $k = 6$. The path is depicted horizontally, the time advances vertically. Each dot represents a request as computed by the recursion. The dots are connected by the Arrow order π_{A}, starting with the root "virtual" request (top-left). The connection between two successive requests illustrates how the Arrow protocol operates: A request sends a message along the diagonal line until it finds the predecessor; the latency and message complexity is represented by the length of the diagonal line, the (cost-free) time a token has to wait for its successor by the length of the vertical line.

4 Conclusions

In this paper we have shown that the Arrow protocol is a distributed ordering algorithm with low overhead, at least in a synchronous message passing model. Since Arrow is an asynchronous protocol it is most natural to ask about its asynchronous complexity. Unfortunately, our analysis does not directly apply for the asynchronous setting. In fact, in our model the trivial D-competitiveness of [1] is already tight in the asynchronous setting. However, if we adopt the model and count time complexity only for Arrow, whereas the optimal offline algorithm is charged time and message complexity, the analysis can be done along the same lines yielding a competitive ratio of $\mathrm{O}(\log D)$ as in the synchronous case.

References

[1] M. J. Demmer and M. Herlihy. The arrow distributed directory protocol. In *International Symposium on Distributed Computing (DISC)*, pages 119–133, 1998.

[2] Y. Emek and D. Peleg. Approximating minimum max-stretch spanning trees on unweighted graphs. In *Proceedings of 15th ACM-SIAM Sympos. Discrete Algorithms (SODA)*, 2004.

[3] D. Ginat, D. Sleator, and R. Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 31(1):3–5, 1989.

[4] M. Herlihy. The aleph toolkit: Support for scalable distributed shared objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, Jan. 1999.

[5] M. Herlihy and S. Tirthapura. Self-stabilizing distributed queueing. In *Proceedings of 15th Interna-*

tional Symposium on Distributed Computing, Oct. 2001.

[6] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive Concurrent Distributed Queueing. In *Proc. of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 127–133, 2001.

[7] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–95, 2001. Also in the Proceedings of the PODC Middleware Symposium, Portland, Oregon, July 2000.

[8] M. Herlihy and M. P. Warres. A tale of two directories: implementing distributed shared objects in Java. *Concurrency: Practice and Experience*, 12(7):555–572, 2000.

[9] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4), 1989.

[10] D. Peleg and E. Reshef. A Variant of the Arrow Distributed Directory Protocol With Low Average Complexity. In *Proc. of the 26th Int. Colloquium on Automata Languages and Programming (ICALP)*, pages 615–624, 1999.

[11] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1), 1989.

[12] R. Rosenkrantz, R. Stearns, and P. Lewis. An Analysis of Several Heuristics for the Traveling Salesman Problem. *SIAM Journal on Computing*, 6(3):563–581, 1977.