Theory Comput. Systems **39**, 875–901 (2006)
DOI: 10.1007/s00224-006-1251-9

Theory of Computing Systems

Dynamic Analysis of the Arrow Distributed Protocol

Maurice Herlihy,[1] Fabian Kuhn,[2] Srikanta Tirthapura,[3] and Roger Wattenhofer[4]

[1]Computer Science Department, Brown University,
Providence, RI 02912, USA
herlihy@cs.brown.edu

[2]Microsoft Research – Silicon Valley,
Mountain View, CA 94043, USA
kuhn@microsoft.com

[3]Department of Electrical and Computer Engineering, Iowa State University,
Ames, IA 50011, USA
snt@iastate.edu

[4]Computer Engineering and Networks Laboratory, ETH Zurich,
CH-8092 Zürich, Switzerland
wattenhofer@tik.ee.ethz.ch

Abstract. Distributed queuing is a fundamental coordination problem that arises in a variety of applications, including distributed directories, totally ordered multicast, and distributed mutual exclusion. The arrow protocol is a solution to distributed queuing that is based on path reversal on a pre-selected spanning tree of the network.

We present a novel and comprehensive competitive analysis of the arrow protocol. We consider the total cost of handling a finite number of queuing requests, which may or may not be issued concurrently, and show that the arrow protocol is $O(s \cdot \log D)$-competitive to the optimal queuing protocol, where s and D are the stretch and the diameter, respectively, of the spanning tree. In addition, we show that our analysis is almost tight by proving that for every spanning tree chosen for execution, the arrow protocol is $\Omega(s \cdot \log(D/s)/\log\log(D/s))$-competitive to the optimal queuing protocol. Our analysis reveals an intriguing connection between the arrow protocol and the nearest neighbor traveling salesperson tour on an appropriately defined graph.

1. Introduction

Ordering of events and messages is at the heart of any distributed system, arising in a multiplicity of applications. Distributed queuing is a fundamental ordering problem, which is useful in many applications ranging from totally ordered multicast to distributed mutual exclusion.

To motivate distributed queuing, consider the problem of synchronizing accesses to a single mobile object in a computer network. This object could be a file that users need exclusive access for writing, or it might just be a privilege, as in the case of distributed mutual exclusion. If a user requests the object which is not on the local node, the request must be transmitted to the current location of the object, and the object should be moved to the user. If there are multiple concurrent requests from users at different nodes, then the requests must be queued in some order, and the object should travel from one user to another down the queue. The main synchronization needed here is the management of the distributed queue. The information needed by every user about the queue is minimal: each user only needs to know the location of the next request in the queue, so that it can pass the object along. Distributed queuing abstracts out the essential part of the above synchronization problem.

In the distributed queuing problem, processors in a message-passing network asynchronously and concurrently request to join a total order (or a distributed queue). The task of the queuing algorithm is to enqueue these requests and extend the total order. Each requesting processor (except for the last request in the queue) should be informed of the identity of its successor in the queue. This is a distributed queue in two senses. Firstly, it can be manipulated by nodes in a distributed system. Secondly, the knowledge of the queue itself is distributed. No single processor, or a small group of processors, needs to have a global view of the queue. Each processor only needs to know its successor in the queue, and thus has a very local view of the queue.

Such a distributed queue can be used in many ways. For example, it can be used in distributed counting by passing an integer counter down the queue, or, as explained earlier, it can be used to ensure mutually exclusive access to a distributed shared object. An efficient implementation of a distributed queue is important for the performance of all these applications.

The *arrow protocol* is an elegant distributed queuing protocol that is based on path reversal on a pre-selected spanning tree of the network. The arrow protocol was invented by Raymond [19] in the context of distributed mutual exclusion, and has since been applied to distributed directories [4], and totally ordered multicast [11]. It has been shown to outperform centralized schemes significantly in practice [12]. However, thus far, there has not been a thorough formal analysis of the arrow protocol. Previous analyses [4], [18] have considered only the sequential case, when there are no concurrent queuing requests, and different requests are always issued far apart in time. Though an analysis of the sequential case gives us some insight into the working of the protocol, one of the most interesting aspects of the arrow protocol is its performance in the concurrent case, when multiple requests are being queued simultaneously.

Our Contribution. We present the first formal performance analysis of the arrow protocol in the presence of concurrent queuing requests. Let s denote the "stretch" of the

pre-selected spanning tree on which the arrow protocol operates, and let D denote the diameter. Informally, the stretch of a tree is the overhead of routing over the tree as opposed to routing over the original network. We provide precise definitions in Section 3.

- We present a competitive analysis showing that the total cost incurred by the arrow protocol to service any finite set of requests, which may or or may not be concurrent, is never more than a factor of $O(s \cdot \log D)$ away from the performance of the "optimal" offline queuing protocol which has omniscient global knowledge and which gets synchronization for free. In other words, the competitive ratio of the arrow protocol is $O(s \cdot \log D)$.
- We also present an almost matching lower bound showing that on any spanning tree of diameter D, the worst-case competitive ratio of the arrow protocol is at least $\Omega(s \cdot \log(D/s)/\log\log(D/s))$. This shows that our upper bound for the competitive ratio is almost tight.

The difficulty in a concurrent analysis of the protocol is as follows. A queuing protocol has many options when faced with concurrent requests. Depending on the origins of the requests, some queuing orders may be much more efficient than others. For example, when presented with simultaneous (or nearly simultaneous) requests from nodes u_1, u_2, and v, where u_1 and u_2 are close to each other but v is far away, it is more efficient to avoid ordering v between u_1 and u_2. The reason is that if v were ordered between u_1 and u_2, information would have to travel between u_1 and v and between v and u_2, which would both lead to long latencies. On the other hand, if v was ordered after (or, equivalently, before) both u_1 and u_2, information would have to travel only between v and u_2 and between u_1 and u_2, which would lead to a lower total latency. More generally, r queuing requests can be ordered in any of $r!$ ways, and depending on the origins of the requests, some orders may be much more efficient than others. Thus, we first need a good characterization of the queuing order of the protocol.

Our analysis employs a novel "nearest-neighbor" characterization of the order in which the arrow protocol queues requests, and this yields a connection between the arrow protocol and the nearest-neighbor heuristic for the traveling salesperson problem (TSP). With the help of this characterization, we derive the competitive ratio of the arrow protocol. In order to establish our main result we must prove a new approximation result for the TSP nearest-neighbor heuristic [20].

This work combines two previous papers [10], [14] and expands on them. Herlihy et al. [10] presented an analysis of the arrow protocol for the *concurrent one-shot case*, when all the queuing requests were issued simultaneously. They showed that the cost of the arrow protocol was always within a factor of $s \cdot \log|R|$ of the optimal, where R is the set of nodes issuing queuing requests, and also provided an almost matching lower bound. Building on this work, Kuhn and Wattenhofer [14] presented an analysis of the more general dynamic case: if nodes are allowed to initiate requests at arbitrary times, the arrow protocol is within a factor of $O(s \cdot \log D)$ of the optimal, and also presented an almost matching lower bound. The present paper extends both previous papers with respect to the used communication model. While in [10] and [14], the arrow protocol is only analyzed for a synchronous communication model, we generalize the analysis for asynchronous communication in this paper.

While the main focus of this paper is the theoretical analysis, we also present results from experiments which corroborate the theoretical results. The experiments show that the performance of the protocol is indeed extremely good in practice, especially under situations of high contention.

1.1. *Previous and Related Work*

The arrow protocol was invented by Raymond [19] in the context of distributed mutual exclusion. Demmer and Herlihy [4] showed that in the sequential case, i.e. when two queuing requests are never simultaneously active, the time and message complexity of any queuing operation was at most D, the diameter of the spanning tree, and the competitive ratio of the arrow protocol was s, the stretch of the pre-selected spanning tree. The protocol has been implemented as a part of the Aleph Toolkit [12]. We have also implemented the protocol, and present our experimental results in Section 5.

Spanning Trees. The arrow protocol runs on a pre-selected spanning tree of the network. Choosing good spanning trees for the protocol is an important problem whose goal is complementary to this paper. While Demmer and Herlihy [4] suggested using a minimum spanning tree, Peleg and Reshef [18] showed that the protocol overhead (at least for the sequential case) is minimized by using a *minimum communication spanning tree* [13]. They further showed that if the probability distribution of the origin of the next queuing operation is known in advance, then it is possible to find a tree whose expected communication overhead for the sequential case is 1.5. They also note that if the adversary (who decides when and where requests occur) is oblivious to the spanning tree chosen, then one can use approximation of metric spaces by tree metrics [1]–[3] to choose a tree whose expected overhead is $O(\log n \log\log n)$ for general graphs, and $O(\log n)$ for constant dimensional Euclidean graphs (the expectation is taken over the coin flips during the selection of the spanning tree). There is a recent breakthrough by Emek and Peleg [6] 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).

Fault-Tolerance. Herlihy and Tirthapura [9] showed that the arrow protocol can be made self-stabilizing [5] with the addition of simple local checking and correction actions.

Other Queuing Protocols. There is another queuing protocol based on path reversal, due to Naimi, Trehel, and Arnold (NTA) [17]. The NTA protocol differs from the arrow protocol in the following significant ways. Firstly, the NTA protocol assumes that the underlying network topology is a completely connected graph, while the arrow protocol does not. Next, the arrow protocol uses a fixed spanning tree, and the pointers can point only to a neighbor in the spanning tree. However, the NTA protocol does not use a fixed spanning tree, and a node's pointer can point to any node in the graph. Thus, in the arrow protocol an ordering operation never travels farther than the diameter of the tree, while in NTA it could travel through every node in the graph. Under certain assumptions

on the probability distribution of operations at nodes, it is shown [17] that an expected $O(\log n)$ messages are required per queuing operation, where n is the number of nodes in the graph. Since we do not assume anything about the probability distribution of operations at nodes, our result is a worst-case result, whereas the analysis of NTA is a probabilistic analysis.

Another queuing protocol is the dynamic distributed object manager protocol by Li and Hudak, as implemented in their Ivy system [15]. As in arrow, Ivy uses pointers to give the way to not-yet collected tokens of previous requests. In contrast to arrow, Ivy needs a complete connection 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. Using this "path shorting" optimization, Ginat et al. [7] proved that the amortized cost of a single request is $\Theta(\log n)$, where n is the number of nodes in the system. However, this analysis is not directly comparable with our analysis of arrow, since the arrow protocol does not assume a complete connection graph, whereas Ivy does.

Roadmap. The remainder of the paper is organized as follows. We first present a description of the arrow protocol in Section 2. In Section 3 we analyze the competitive ratio of the arrow protocol. In Section 4 we present a lower bound on the competitive ratio. In Section 5 we present experimental results of our protocol implementation.

2. The Arrow Protocol

The purpose of any queuing protocol is to order operations totally. The information returned by a queuing protocol is as follows. For each operation issued by a node, the node should be informed of the successor of the operation, except for the following case: if there is currently no successor to an operation (it is the globally last element in the queue, and no more operations are ordered after it), then the issuing node is not informed anything. Queuing operations can be issued by nodes asynchronously, and the same node might issue many operations. Consider a queuing operation a issued by node v. Suppose a is ordered behind operation b. The queuing of a is considered complete when the node which issued operation b is informed that b's successor is a.

We begin with an informal description of the arrow protocol. The protocol runs on a pre-selected spanning tree of the network. Initially, some node in the tree contains the tail of the queue; we call this the *root*. Every node in the network has a pointer which points to a neighbor in the tree. The pointers are initialized such that following the chain of pointers starting from any node leads to the root.

When node v issues a queuing operation, the operation follows the chain of pointers starting from v towards the root, simultaneously flipping the pointers on the way, back towards v. Once v's operation reaches the root, it has found its predecessor, and its queuing is complete. The queue has been extended, and the tail of the queue has moved to node v. The simple action of flipping only those pointers on the path to the root has modified the global state such that following the chain of pointers from any node now leads to the new root, v. A new queuing operation from another node w will now be queued behind v's operation.

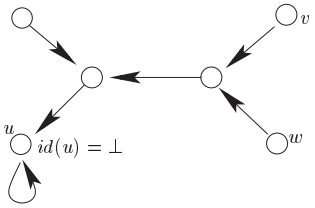

Fig. 1. Arrow protocol: initial system state. The pointers point to neighbors in the tree, leading to a unique sink u.

More formally, we model the network as a graph $G = (V, E)$, where V is the set of processors and E the set of point-to-point FIFO communication links between processors. The protocol chooses a spanning tree T of G. Each node $v \in V$ has a pointer, denoted by $link(v)$, which is either a neighboring node in the spanning tree, or v itself. Each node v also has an attribute $id(v)$, which is the unique identifier of the previous queuing operation issued by v. If v has not issued any queuing operations so far, then $id(v)$ is the special symbol $\perp$. A node v is called a *sink* if $link(v) = v$. The *link* pointers are initialized so that following the pointers from any node leads us to a unique sink, the root of the tree; an example is shown in Figure 1.

In the arrow protocol, when a node v initiates a queuing operation whose *id* is a, it executes the following sequence of steps atomically:

- Set $id(v) \leftarrow a$.
- Send $queue(a)$ message to $u_1 = link(v)$.
- Set $link(v) \leftarrow v$.

When node u_i receives a $queue(a)$ message from node u_{i-1}, it executes the following atomic sequence of steps, called a *path reversal*. Let $u_{i+1} = link(u_i)$:

- Flip u_i's link, i.e. set $link(u_i) \leftarrow u_{i-1}$.
- If $u_{i+1} \neq u_i$, then forward message $queue(a)$ to u_{i+1}.
- If $u_{i+1} = u_i$, then operation a has been queued behind $id(u_i)$. The queuing of a is considered complete since u_i has been informed of the identity of the successor of operation $id(u_i)$.

In some applications, additional messages need to be sent. For example, in synchronizing accesses to mobile objects, it is necessary for u_i to send the actual object to v through a message. However, we do not consider these additional messages as a part of the queuing protocol itself.

Thus far, we have described the protocol as if the operations were being executed sequentially, spaced far apart in time. The striking feature is that the protocol works just as well even in the case of concurrent queuing operations. An example execution with two concurrent queuing operations is illustrated in Figures 1–5. For a proof of correctness, we refer the reader to [4], where the authors argue that every concurrent execution of the protocol is equivalent to some sequential execution, and since every sequential execution is correct, so is every concurrent execution.

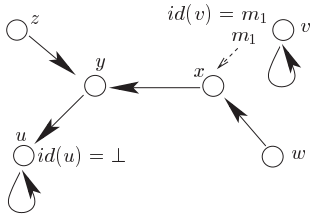

Fig. 2. Arrow protocol step 1. Node v initiates a queuing request, and sends message m_1, on its way to x.

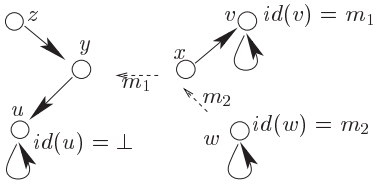

Fig. 3. Arrow protocol step 2. Node w initiates another request, and sends m_2, now on its way to x.

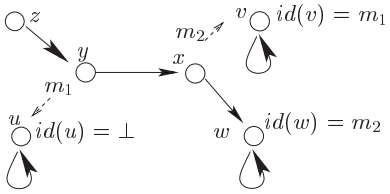

Fig. 4. Arrow protocol step 3. Messages m_1 and m_2 follow the pointers, reversing their directions along the way. Note that m_2 has been deflected towards v.

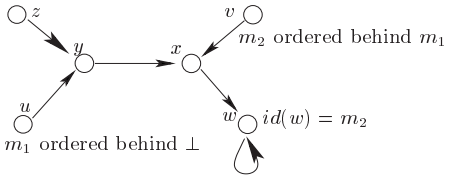

Fig. 5. Arrow protocol step 4. Both m_1 and m_2 find their predecessors in the total order and are queued concurrently.

3. Analysis

We now describe our analysis of the cost of the arrow protocol under concurrent access to the queue. Our analysis is organized as follows. We first define our model and the cost metrics more precisely. In Section 3.4 we give a characterization of the queuing order of the arrow protocol, which will help us derive the upper bound on its cost. This is followed by an analysis which bounds the cost of an optimal algorithm with a Manhattan Traveling Salesperson Tour from below. The optimal algorithm pays nothing for synchronization, and can order the requests differently from the arrow protocol to minimize the cost. In Section 3.6 we give a new analysis of the TSP nearest neighbor heuristic which will allow us to derive the competitive ratio in Sections 3.7 and 3.8.

3.1. *Model*

We are given a graph $G = (V, E)$ representing the network, and a spanning tree T of the graph. We first consider a synchronous model of computation, where the latency of a communication link is predictable. In particular, we focus on the case where every edge has unit latency. In Section 3.8 we extend our results for synchronous communication to asynchronous systems. For nodes $u, v \in V$, let $d_T(u, v)$ denote the distance between u and v on T, and let $d_G(u, v)$ denote the distance between u and v on the graph G.

Definition 3.1. Given a graph $G = (V, E)$ and a spanning tree T, the stretch of T is defined as $s := \max_{u,v \in V} d_T(u, v)/d_G(u, v)$.

A *queue*() message arriving at a node is processed immediately, and simultaneously arriving messages are processed in an arbitrary order. Our analysis holds irrespective of the order in which the *queue*() messages are locally processed. This assumes that node v can process up to d_v messages in a time step at a node, where d_v is v's degree. Because in practice the time needed to service a message is small when compared with communication latency, this assumption seems reasonable. For a simple example with concurrent *queue*() operations, see Figure 6.

In summary, the optimal queuing algorithm has the following additional power over the arrow protocol. First, it can globally order queuing requests differently from

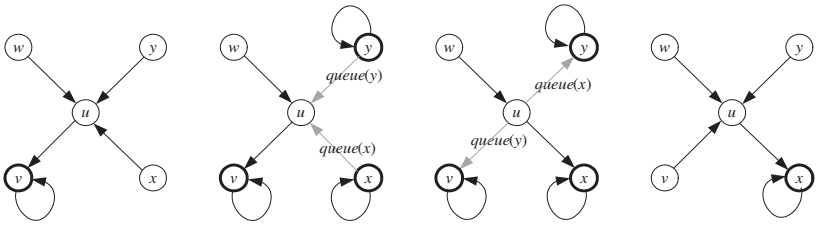

Fig. 6. Arrow protocol: concurrent *queue* messages. Initially node v is selected as the tail of the queue. Nodes x and y both choose to join the queue simultaneously. Message *queue*(y) arrives at node u before *queue*(x). Finally, *queue*(x) and *queue*(y) find their respective predecessors y and v in the queue, and x is the new tail of the queue.

the arrow protocol, since it has complete knowledge of current and future requests. Next, at every single node, the optimal algorithm can locally order arriving messages differently from the arrow protocol. Finally, the optimal algorithm can communicate over the communication graph G while the arrow protocol has to communicate over the spanning tree T.

The Concurrent Queuing Setting. Each queuing request is an ordered pair (v, t) where $v \in V$ is the node where the request was issued, and $t \geq 0$ is the time when it was issued. Let $R = \{r_0 = (v_0, t_0), r_1 = (v_1, t_1), \ldots\}$ denote the set of all queuing requests. The requests r_i in R are indexed in the order of non-decreasing time, with ties broken arbitrarily, so that $i < j \Longrightarrow t_i \leq t_j$. Note that this tie-breaking rule is not used in any way in the algorithm, and is just a convenient way for indexing the requests.

Suppose a request $r = (v, t)$ is queued behind another request $r' = (v', t')$. The queuing of r is considered complete at the time v' is informed that the successor of r' is r.

Definition 3.2. If request $r = (v, t)$ is queued behind $r' = (v', t')$, then the latency r is the time that elapses between the initiation of the request (i.e. t) and the time v' is informed that the successor of r' is r.

Definition 3.3. The cost of any queuing algorithm is the total latency, which is the sum of the latencies of all the individual queuing requests.

The reason for using the above metric for latency is as follows. In many applications, the only knowledge needed about the distributed queue is the identity of the successor of a node's request. For example, in synchronizing accesses to a mobile object, each node only needs to know where to send the object next, so knowledge of the successor suffices. If v also needs to know the identity of the predecessor of request r then v' can send v a message, and the additional delay to do so will not be more than the above defined latency. The cost in such a case would be comparable with our current definition of latency.

Another option would be to consider the total message complexity (number of messages sent) as the cost metric and to ignore latency. However, this does not work for an online algorithm for the following reason. Consider a scenario where only two nodes u, v initiate requests. An optimal offline algorithm may order every request of u before any request 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 (which experiences a huge latency, for that matter). No online algorithm can compete against such a powerful adversary.

3.2. *Cost of Arrow*

We first look at the cost of the arrow algorithm. Suppose the arrow protocol runs and orders all the requests in R into a queue. Let π_{A} be the resulting queuing order, i.e. $\pi_{\mathrm{A}}(i)$ denotes the index of the ith request in arrow's order. We introduce $r_0 = (\text{root}, 0)$ representing the "virtual" request at the root, which is the start of the queue; since this request should be the first in any queuing order, including the one induced by arrow, we have $r_{\pi_{\mathrm{A}}(0)} = r_0$.

As already proved in [4], in arrow each request r_j will send a message to its predecessor r_j using the direct path in the spanning tree.

Therefore, if the arrow protocol orders request r_i immediately after request r_j, then the latency of r_i using the arrow protocol (denoted by $c_{\mathrm{A}}(r_i, r_j)$) is given by

$$c_{\mathrm{A}}(r_i, r_j) = d_T(v_i, v_j). \tag{1}$$

According to Definition 3.3, the total latency of the arrow algorithm for the request set R, denoted by cost^R_{arrow} is

$$\mathrm{cost}^R_{arrow} = \sum_{i=1}^{|R|} c_{\mathrm{A}}(r_{\pi_{\mathrm{A}}(i)}, r_{\pi_{\mathrm{A}}(i-1)}) = \sum_{i=1}^{|R|} d_T(v_{\pi_{\mathrm{A}}(i)}, v_{\pi_{\mathrm{A}}(i-1)}). \tag{2}$$

3.3. *Cost of an Optimal Offline Algorithm*

We now look at the cost of an optimal offline ordering algorithm *Opt* that has complete knowledge about all the requests R in order to determine the queuing order. We assume that the knowledge of R can only be used to determine the queuing order and to send the right messages. It does not make sense to assume that all nodes know R from the beginning because in this case, nodes know their requests' successors without communicating. Also for *Opt*, requests occur distributedly and dynamically. Hence, a node v does not know about a request $r = (v, t)$ before time t. All other nodes can only know about r if they are informed by v. Thus, if r is the successor of a request $r' = (v', t')$, v' cannot know about r before time $t + d_G(v, v')$. Let π_{O} be the queuing order induced by *Opt*. Suppose that *Opt* ordered r_j immediately after r_i. See Figure 7.

From Definition 3.2, the latency of r_j is the time elapsed between t_j and the instant when v_i is informed that the successor of r_i is r_j. The following conditions place lower bounds on the latency of r_j:

- At time t_j, only v_j knows about request r_j. Node v_i cannot know about the existence of r_j before time $t_j + d_G(v_i, v_j)$. Thus, the latency of r_j in *Opt* must be at least $d_G(v_i, v_j)$.

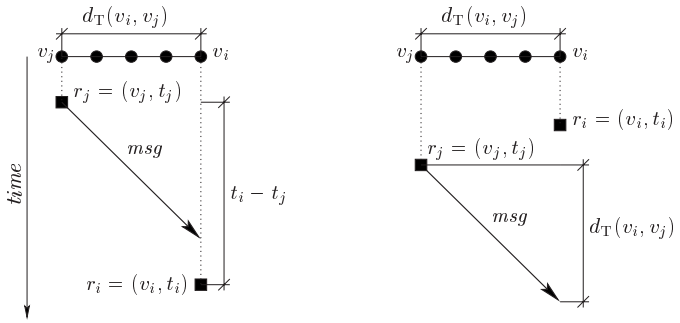

Fig. 7. The latency of an optimal algorithm for ordering r_j after r_i. Left: v_i is informed about r_j before time t_i, therefore the latency is $t_i - t_j$. Right: Request r_i has been issued at v_i when the message arrives from v_j. Thus the latency of r_j is $d_G(v_i, v_j)$.

- Since request r_i does not exist before t_i, v_i cannot be informed of r_i's successor before t_i. Thus, the latency of r_j's request is at least $t_i - t_j$.

Let $c_{Opt}(r_i, r_j)$ and $c_{\mathrm{O}}(r_i, r_j)$ be defined as follows:

$$c_{Opt}(r_i, r_j) := \max\{d_G(v_i, v_j), t_i - t_j\},$$
$$c_{\mathrm{O}}(r_i, r_j) := \max\{d_T(v_i, v_j), t_i - t_j\} \leq s \cdot c_{Opt}(r_i, r_j), \tag{3}$$

where $s := \max_{(u,v)} d_G(u, v)/d_T(u, v)$ is the stretch of the spanning tree T. From the above argument, we have the following fact:

Fact 3.4. *The latency of r_j in the optimal algorithm, if it is ordered immediately after r_i, is at least $c_{Opt}(r_i, r_j) \geq c_{\mathrm{O}}(r_i, r_j)/s$.*

Let cost^R_{Opt} denote the total latency (defined in Definition 3.3) of *Opt* for request set R. Due to Fact 3.4, we have

$$\mathrm{cost}^R_{Opt} \geq \min_{\pi} \sum_{i=1}^{|R|} c_{Opt}(r_{\pi(i)}, r_{\pi(i-1)}) \geq \frac{1}{s} \cdot \min_{\pi} \sum_{i=1}^{|R|} c_{\mathrm{O}}(r_{\pi(i)}, r_{\pi(i-1)}). \tag{4}$$

In the above relation, the minimum is taken over all possible permutations π of requests in R. Let π_{O} denote the order which minimizes the right-hand side sum of (4).

The competitive ratio ρ achieved by the arrow algorithm is the worst case ratio between the cost of arrow and the cost of an optimal offline ordering strategy, the worst case being taken over all possible request sets R:

$$\rho := \min_{R} \frac{\mathrm{cost}^R_{arrow}}{\mathrm{cost}^R_{Opt}}. \tag{5}$$

3.4. *The Arrow Protocol in the Dynamic Setting*

We now take a closer look at the ordering produced by the arrow algorithm. We will define a new cost measure c_T on the set R, and show that π_{A}, the ordering produced by arrow, corresponds to a nearest-neighbor TSP path with respect to this cost, starting from the root request. Then, using amortized analysis, we will show that this new cost c_T is comparable with latency cost c_{A}.

Definition 3.5. The cost c_T is defined over requests in R as follows. For $r_i, r_j \in R$, let $d = t_j - t_i + d_T(v_i, v_j)$. If $d \geq 0$, then $c_{\mathrm{T}}(r_i, r_j) = d$. If $d < 0$, then $c_{\mathrm{T}}(r_i, r_j) = t_i - t_j + d_T(v_i, v_j)$.

Note that c_T is asymmetric: $c_{\mathrm{T}}(r_i, r_j)$ does not necessarily equal $c_{\mathrm{T}}(r_j, r_i)$. We will need the following fact which follows from the definition of c_T.

Fact 3.6. *For all $r_i, r_j \in R$, $c_{\mathrm{T}}(r_i, r_j) \geq 0$.*

A nearest-neighbor TSP path on R induced by cost c is a traveling salesperson path which starts from the root $r_0 = (\mathrm{root}, 0)$ and visits the requests of R in an order π_{NN} that

satisfies the following constraints:

$$c(r_0, r_{\pi_{\mathrm{NN}}(1)}) = \min_{r \in R} c(r_0, r), \tag{6}$$

$$c(r_{\pi_{\mathrm{NN}}(i)}, r_{\pi_{\mathrm{NN}}(i+1)}) = \min_{(r \in R) \wedge (r \notin \{r_{\pi_{\mathrm{NN}}(0)} \cdots r_{\pi_{\mathrm{NN}}(i-1)}\})} c(r_{\pi_{\mathrm{NN}}(i)}, r). \tag{7}$$

For $0 \leq i \leq |R| - 1$, we define the following:

- R_i is the subset of requests $\{r_{\pi_{\mathrm{A}}(j)} \in R \mid j > i\}$.
- F_i is a configuration where all arrows on the spanning tree point towards node $v_{\pi_{\mathrm{A}}(i)}$. Note that in the initial configuration F_0, all arrows are pointing towards the root, $v_{\pi_{\mathrm{A}}(0)}$.
- E_i is an execution of the arrow protocol starting from configuration F_i with all requests in R_i being issued.

Lemma 3.7. *For each $1 \leq i \leq |R| - 1$, no request in R_{i-1} except for $r_{\pi_{\mathrm{A}}(i)}$ will be able to distinguish locally between executions E_{i-1} and E_i.*

Proof. Executions E_{i-1} and E_i start from F_{i-1} and F_i respectively. Configurations F_{i-1} and F_i are equivalent, except that all pointers on the path P between $v_{\pi_{\mathrm{A}}(i-1)}$ and $v_{\pi_{\mathrm{A}}(i)}$ are pointing towards $v_{\pi_{\mathrm{A}}(i-1)}$ in F_{i-1} and towards $v_{\pi_{\mathrm{A}}(i)}$ in F_i (see Figure 8). Among requests in R_{i-1}, $r_{\pi_{\mathrm{A}}(i)}$ is the next request in the total order. Thus $r_{\pi_{\mathrm{A}}(i)}$ will take a direct path from $v_{\pi_{\mathrm{A}}(i)}$ to $v_{\pi_{\mathrm{A}}(i-1)}$ on the tree.

Assume, for the sake of contradiction, that there is a request $r_{\pi_{\mathrm{A}}(j)}$ with $j > i$ (i.e. $r_{\pi_{\mathrm{A}}(j)}$ is ordered after $r_{\pi_{\mathrm{A}}(i)}$) that is able to distinguish between E_{i-1} and E_i. Then $r_{\pi_{\mathrm{A}}(j)}$ must have been able to see an arrow on P pointing towards $v_{\pi_{\mathrm{A}}(i-1)}$ before $r_{\pi_{\mathrm{A}}(i)}$ reversed it. In such a case, $r_{\pi_{\mathrm{A}}(j)}$ would have deflected $r_{\pi_{\mathrm{A}}(i)}$, and would have been ordered before $r_{\pi_{\mathrm{A}}(i)}$. This contradicts the assumption that $r_{\pi_{\mathrm{A}}(i)}$ comes earlier than $r_{\pi_{\mathrm{A}}(j)}$ in the total order. □

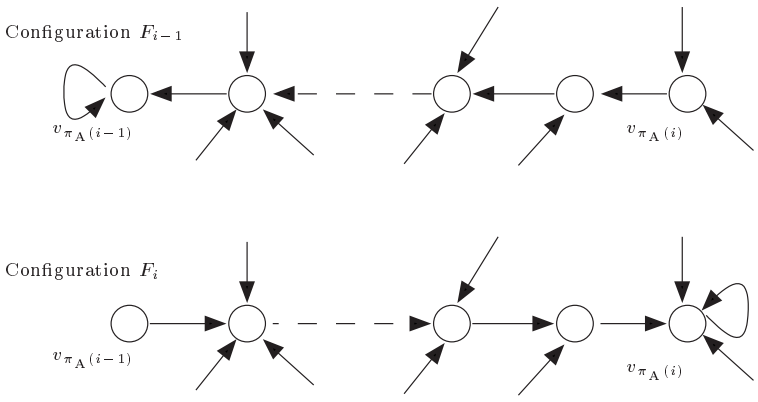

Fig. 8. The executions starting from F_{i-1} and F_i are indistinguishable to every request but $r_{\pi_{\mathrm{A}}(i)}$ (issued by $v_{\pi_{\mathrm{A}}(i)}$).

Lemma 3.8. *The queuing order of arrow, π_A is a nearest-neighbor TSP path on R induced by the cost c_T, starting with the dummy token/request $r_0 = (\text{root}, 0)$.*

Proof. We first prove (6) for π_A. Let S be the set of requests which minimize $c_T(r_0, s)$, for $s \in S$. Since the start time of r_0 is 0, $c_T(r_0, r_i) = t_i + d_T(\text{root}, v_i)$.

When initiated, the requests of S start moving towards root traveling on the tree T, since the tree is initialized with all arrows pointing towards root. Note that at each point in time, all already initiated requests in S are at the same distance from root.

If two or more find requests from S meet at a node, only one continues towards root, and the others are deflected. Thus, at least one request from S arrives at the root, and no request outside of S can make it to root before that. It follows that the immediate successor of r_0 in π_A is a request from S, thus proving (6) for the ordering π_A using metric c_T.

Suppose (7) was true for $i = k$ for π_A using metric c_T. We will now show this to be true for $i = k + 1$, i.e. that the request succeeding $r_{\pi_A(k+1)}$ is that request in R_{k+1} that is closest to $r_{\pi_A(k+1)}$ according to metric c_T.

Consider executions E_k and E_{k+1}. From Lemma 3.7, no request in R_k except for $r_{\pi_A(k+1)}$ will be able to distinguish between the two executions. For the purpose of ascertaining a request succeeding $r_{\pi_A(k+1)}$ in execution E_k, it is sufficient to consider the execution E_{k+1} where all arrows are pointing to $v_{\pi_A(k+1)}$ and all requests in R_{k+1} are issued. From the previous argument, the next request in this total order, $r_{\pi_A(k+2)}$, is the request in R_{k+1} that is closest to $r_{\pi_A(k+1)}$ according to metric c_T. Thus, (7) is also true for $i = k + 1$. □

Lemma 3.9. *Consider any two requests $r_i = (v_i, t_i) \in R$ and $r_j = (v_j, t_j) \in R$. If $t_j - t_i > d_T(v_i, v_j)$, then r_i is ordered before r_j by arrow.*

Proof. Proof by contradiction. Suppose r_j was ordered before r_i. Say $r_k = (v_k, t_k)$ was the immediate predecessor of r_j in the order formed by the arrow protocol.

From Lemma 3.8, we have the following:

$$
\begin{aligned}
&c_T(r_k, r_j) \le c_T(r_k, r_i),\\
&t_j - t_k + d_T(v_k, v_j) \le t_i - t_k + d_T(v_k, v_i),\\
&(t_j - t_i) \le d_T(v_k, v_i) - d_T(v_k, v_j),\\
&\qquad \le d_T(v_j, v_i).
\end{aligned}
$$

The last inequality is due to the triangle inequality for the metric $d_T(u, v)$. This contradicts our initial assumption, and completes the proof. □

Recall that π_A denotes the order induced by the arrow protocol over the request set R, and let $r_{\pi_A(0)}$ denote the root request (root, 0). Recall that cost^R_{arrow} denotes the cost of the arrow protocol over the request set R.

Lemma 3.10. *Let*

$$C_T = \sum_{i=0}^{|R|-1} c_T(r_{\pi_A(i)}, r_{\pi_A(i+1)}).$$

Then

$$\text{cost}^R_{arrow} = C_T + t_{\pi_A(|R|)}.$$

Proof. From the proof of Lemma 3.8, we know that

$$c_T(r_{\pi_A(i)}, r_{\pi_A(i+1)}) = t_{\pi_A(i+1)} - t_{\pi_A(i)} + d_T(v_{\pi_A(i)}, v_{\pi_A(i+1)}).$$

Thus,

$$\begin{aligned} C_T &= \sum_{i=0}^{|R|-1} c_T(r_{\pi_A(i)}, r_{\pi_A(i+1)}) \\ &= \sum_{i=0}^{|R|-1} (t_{\pi_A(i+1)} - t_{\pi_A(i)} + d_T(v_{\pi_A(i)}, v_{\pi_A(i+1)})) \\ &= t_{\pi_A(|R|)} + \sum_{i=0}^{|R|-1} d_T(v_{\pi_A(i)}, v_{\pi_A(i+1)}). \end{aligned}$$

From (2), we know that

$$\text{cost}^R_{arrow} = \sum_{i=1}^{|R|} d_T(v_{\pi_A(i)}, v_{\pi_A(i-1)}).$$

The lemma follows. □

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 most significant 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 3.11 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 either arrow or the offline algorithm.

Notation. Let $R_{\leq t} = \{r_i \in R \mid t_i \leq t\}$ and $R_{\geq t} = \{r_i \in R \mid t_i \geq t\}$.

Lemma 3.11. *Let $r_i = (v_i, t_i)$ and $r_{i+1} = (v_{i+1}, t_{i+1})$ be two consecutive requests with respect to times of occurrence. Further choose two requests $r_a \in R_{\leq t_i}$ and $r_b \in R_{\geq t_{i+1}}$ minimizing $\delta := t_b - t_a - d_T(v_a, v_b)$. If $\delta > 0$, every request $r = (v, t) \in R_{\geq t_{i+1}}$ 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 3.9, the requests in $R_{\leq t_i}$ are ordered before the requests in $R_{\geq t_{i+1}}$ by arrow. By the definition of δ, this does not change is we replaced requests as above. The transformation therefore does not change the ordering (due to arrow) of the requests in $R_{\leq t_i}$.

Let r be the latest 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 requests 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 optimal cost $c_{\mathrm{O}}(r, r')$ between any two requests $r = (v, t)$ and $r' = (v', t')$ cannot be increased by the transformation. If either (1) both r and r' are in $R_{\leq t_i}$ or (2) both r and r' are in $R_{\geq t_{i+1}}$, then $c_{\mathrm{O}}(r, r')$ does not change.

If $r \in R_{\geq t_{i+1}}$ and $r' \in R_{\leq t_i}$, then by the definition of δ, $c_{\mathrm{O}}(r, r')$ is reduced by δ. If $r \in R_{\leq t_i}$ and $r' \in R_{\geq t_{i+1}}$, the term $\max\{0, t - t' + d_T(v, v')\}$ remains zero before and after the transformation. From the definition of c_{O} (equation (3)) $c_{\mathrm{O}}(r, r')$ remains $d_T(v, v')$ before and after the transformation. □

In the following we assume that all requests are already transformed according to Lemma 3.11.

Lemma 3.12. *Let $r_i = (v_i, t_i)$ and $r_{i+1} = (v_{i+1}, t_{i+1})$ be two consecutive requests with respect to 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 $d_T(r_a, r_b) \geq t_b - t_a$.*

Proof. If it is not the case, we can apply the transformation of Lemma 3.11. □

Lemma 3.13. *The cost $c_{\mathrm{T}}(r_i, r_j)$ of the longest edge (r_i, r_j) on arrow's path 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 3.12, we can assume that the temporal difference between two successive requests (with respect to 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

$$t_k - t_i \geq D + \varepsilon > d_T(v_i, v_k)$$

and therefore, by Lemma 3.9, 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

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

□

3.5. *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 3.14 (Manhattan Metric). The Manhattan metric $c_{\mathrm{M}}(r_i, r_j)$ is defined as

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

Lemma 3.15. *Let π be an ordering and let 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*

$$C_{\mathrm{M}} \leq 4C_{\mathrm{O}} + t_{\pi(|R|)}.$$

Proof. We can lower bound the optimal cost of (3) by

$$c_{\mathrm{O}}(r_i, r_j) = \max\{d_T(r_i, r_j), t_i - t_j\} \geq \frac{d_T(r_i, r_j) + \max\{0, t_i - t_j\}}{2}. \tag{8}$$

Let $D_T := \sum_{i=1}^{|R|} d_T(v_{\pi(i-1)}, v_{\pi(i)})$ and $T_U := \sum_{i=1}^{|R|} \max\{0, t_{\pi(i-1)} - t_{\pi(i)}\}$, that is, we have $C_{\mathrm{O}} \geq (D_T + T_U)/2$. Further, let $T_U := \sum_{i=1}^{|R|} \max\{0, t_{\pi(i-1)} - t_{\pi(i)}\}$ and let $T := \sum_{i=1}^{|R|} |t_{\pi(i)} - t_{\pi(i-1)}|$. We clearly have $T = T_U + T_D$. Because $t_{\pi(0)} = t_0 = 0$, we also have $T_D = T_U + t_{\pi(|R|)}$. Adding T_U on both sides yields $T = 2T_U + t_{\pi(|R|)}$ and therefore

$$2\sum_{i=1}^{|R|} \max\{0, t_{\pi(i-1)} - t_{\pi(i)}\} = \sum_{i=1}^{|R|} |t_{\pi(i)} - t_{\pi(i-1)}| - t_{\pi(|R|)}.$$

We thus have

$$\begin{aligned} 4C_{\mathrm{O}} &\geq 2D_T + 2\sum_{i=1}^{|R|} \max\{0, t_{\pi(i-1)} - t_{\pi(i)}\} \\ &\geq D_T + \sum_{i=1}^{|R|} |t_{\pi(i)} - t_{\pi(i-1)}| - t_{\pi(|R|)} \\ &= C_{\mathrm{M}} - t_{\pi(|R|)}. \end{aligned}$$

The last equation follows from the definition of C_{M}. □

Lemma 3.16. *Let π be an order and let C_{M} be the Manhattan cost for ordering all requests in order π. We have*

$$C_{\mathrm{M}} \geq \tfrac{3}{2} t_{|R|},$$

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.

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

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 (with respect to 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 first request occurring 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 $d_T(v_a, v_b)$ 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, that is, $c_{\mathrm{M}} = c_{\mathrm{M}_d} + c_{\mathrm{M}_t}$. To get a bound on c_{M_t}, we define $\Delta t_i^{(1)}$ and $\Delta t_i^{(2)}$ as follows:

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

By the definition of the R_i, we have

$$c_{\mathrm{M}_t} \geq 2 \sum_{i=1}^{k} \Delta t_i^{(2)} + \sum_{i=1}^{k-1} \Delta t_i^{(1)}. \tag{9}$$

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 $t_b - t_a \leq d_T(v_a, v_b)$. Assume that $r_a \in R_i$ and $r_b \in R_j$ for $j > i$. Further, let $c_{\mathrm{M}_d}(i, j)$ 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

$$c_{\mathrm{M}_d}(i, j) \geq d_T(v_a, v_b) \geq \sum_{\ell=i}^{j-1} \Delta t_\ell^{(1)}. \tag{10}$$

By Lemma 3.12, we can assume that 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 $t_b - t_a \leq d_T(v_a, v_b)$. 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 3.12, 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 (10) for all pairs $r_{a,i}$ and $r_{b,i}$, each edge is at most counted twice and therefore

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

Combining this with (9) concludes the proof. □

Lemma 3.17. *Let π be an order and let 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*

$$C_{\mathrm{M}} \leq 12 C_{\mathrm{O}}.$$

Proof. By the Lemmas 3.15 and 3.16, we have

$$\tfrac{3}{2} t_{|R|} \leq 4 C_{\mathrm{O}} + t_{|R|}$$

and therefore $t_{|R|} \leq 8 C_{\mathrm{O}}$. (Note that $t_{|R|} \geq t_{\pi(|R|)}$.) Applying this to Lemma 3.15 completes the proof. □

3.6. *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. Rosenkrantz et al. [20] have shown that the cost of a nearest-neighbor TSP tour is always within a factor $\log N$ of the cost of an optimal TSP tour on a graph with N nodes, for which the distance metric obeys the triangle inequality. 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, the triangle inequality is a necessary condition for the analysis of [20]. Here, we give a stronger and more general approximation ratio for the nearest-neighbor heuristic. Instead of the triangle inequality, we have a cost function which is upper bounded by a metric for which the triangle inequality holds.

Theorem 3.18. *Let V be a set of $N := |V|$ nodes and let $d_n\colon V \times V \to \mathbb{R}$ and $d_o\colon V \times V \to \mathbb{R}$ be distance functions between nodes of V. For d_n and d_o, the following*

conditions hold:

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

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

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

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 of 2. We show that for each class the sum of the lengths of the edges is at most $\frac{3}{2} \cdot C_O$. We therefore look at a single class $\mathcal{C}$ of edges. Let d be the length of the shortest edge (with respect to 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 $\mathcal{C}$ with the length (with respect to 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. Without loss of generality, 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 endpoint 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 three times the sum of the $\lceil |\mathcal{C}|/2 \rceil$ smallest edges of $\mathcal{C}$. This completes the proof. □

3.7. *Complexity of Arrow*

In this section we prove the competitiveness of arrow by putting our individual parts together.

Theorem 3.19. *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*

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

where s and D are the stretch and the diameter of the spanning tree T, respectively.

Proof. We first show that

$$C_T \leq 3\lceil \log_2(3D) \rceil \cdot C_{\mathrm{M}}. \tag{11}$$

Equation (11) can be derived from the TSP NN result of Theorem 3.18 as follows. c_T and c_{M} comply with the conditions for $d_n(u, v)$ and $d_o(u, v)$, respectively. By Lemma 3.8, $c_T \geq 0$. Further, by the definition of c_T, we have

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

Clearly, the triangle inequality holds for the Manhattan metric c_{M}. The only thing missing to apply Theorem 3.18 is abound on the ratio of the longest and the shortest edge on arrow's NN path. By Lemma 3.13, 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). Theorem 3.18 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 at most an additional factor of 2.

By applying Lemmas 3.10 and 3.17, the theorem can now be derived as follows:

$$\begin{aligned} \mathrm{cost}^R_{arrow} &= C_T + t_{\pi_{\mathrm{A}}|R|} \leq C_T + C_{\mathrm{M}} \\ &\leq \left(3\lceil \log_2(3D) \rceil + 1\right) \cdot C_{\mathrm{M}} \in O(s \cdot \log D \cdot \mathrm{cost}^R_{Opt}). \end{aligned}$$

□

3.8. *Complexity of Arrow in the Asynchronous Model*

One of the major reasons for using the arrow protocol is its correctness under arbitrary concurrency in a completely asynchronous environment [4]. So far we have simplified this general setting by considering a synchronous system where the delay of each message is exactly 1. In this section we show that Theorem 3.19 also holds when assuming an asynchronous communication model.

In an asynchronous message passing model, message delays are not bounded by a constant, that is, messages can be arbitrarily fast or slow. However, all messages arrive at their destinations after a finite amount of time. To have a notion of time in asynchronous systems, it is commonly assumed for the analysis that each message has a delay of at most one time unit. The time complexity is then defined as the worst-case (every possible execution) running time, assuming that each message occurs a delay of at most one time unit.

We first look at the cost of an optimal asynchronous queuing algorithm. Clearly, the synchronous time complexity of an algorithm is a lower bound on the asynchronous time complexity. If we assume also that an optimal algorithm has to cope with worst-case message delays, we can therefore use the synchronous cost of *Opt* given by Lemma 3.17 as a lower bound for the cost of *Opt* in the asynchronous model.

The asynchronous cost of the arrow protocol is defined by Definitions 3.2 and 3.3 as for the synchronous case. That is, the latency cost for queuing a request $r_j = (v_j, t_j)$ after $r_i = (r_i, t_i)$ is the time from r_j's initiation (time t_j) until v_i receives the message from v_j. In contrast to a synchronous system, this time does not only depend on t_i, t_j, and $d_T(v_i, v_j)$ but also on the message delays when sending the request from v_j to v_i.

Let us consider an asynchronous execution of the arrow protocol for a given request set R. To analyze the given execution, we can assume that the message delays are scaled such that the latency of the slowest message between adjacent nodes is 1. Let π'_{A} be the ordering of the requests resulting from this execution. For two consecutive (with respect to π'_{A}) requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$, we define $c'_{\mathrm{A}}(r_i, r_j)$ to be the time between the occurrence of r_j at time t_j and v_i being informed about r_j. Let $\mathrm{cost}'^{R}_{Arrow}$ be the total cost of the given arrow execution. By Definitions 3.2 and 3.3 and because the largest message delay is assumed to be 1, we have

$$\mathrm{cost}'^{R}_{Arrow} := \sum_{i=1}^{|R|} c'_{\mathrm{A}}(r_{\pi'_{\mathrm{A}}(i-1)}, r_{\pi'_{\mathrm{A}}(i)}) \leq \sum_{i=1}^{|R|} d_T(v_{\pi'_{\mathrm{A}}(i-1)}, v_{\pi'_{\mathrm{A}}(i)}).$$

Analogously to cost c_T from Section 3.4, we define a cost measure c'_T as follows:

$$c'_T(r_i, r_j) := \begin{cases} t_j - t_i + c'_A(r_i, r_j) & \text{if } r_j \text{ is the request directly after } r_i \text{ with respect to } \pi'_{\mathrm{A}} \\ c_T(r_i, r_j) & \text{otherwise.} \end{cases}$$

Analogous to the synchronous case, for requests $r_{\pi'_{\mathrm{A}}(i-1)}$ and $r_{\pi'_{\mathrm{A}}(i)}$ which are consecutive with respect to π'_{A}, $c'_T(r_i, r_j)$ is the time difference between the initiation of r_i and v_i being informed about r_j. Hence, since $c_T(r_i, r_j) \geq 0$ (Fact 3.6), we also have $c'_T(r_i, r_j) \geq 0$ and because $c'_{\mathrm{A}}(r_i, r_j) \leq d_T(v_i, v_j)$, we get $c'_T(r_i, r_j) \leq c_T(r_i, r_j)$ and therefore

$$0 \leq c'_T(r_i, r_j) \leq c_T(r_i, r_j) \leq c_{\mathrm{M}}(r_i, r_j). \tag{12}$$

In analogy to Lemma 3.8 for the synchronous case, we obtain the following lemma.

Lemma 3.20. *The queuing order π'_{A} resulting from a given asynchronous execution of the arrow protocol is an NN TSP path on R induced by the cost c'_T, starting with the dummy token/request $r_0 = (\mathrm{root}, 0)$.*

Proof. The proof of Lemma 3.7 does not make use of any of the special properties of synchronous systems. Therefore, Lemma 3.7 also holds in the asynchronous setting (when replacing π_{A} by π'_{A} in the definitions of R_i, F_i, and E_i). It therefore suffices to prove that (6) holds for π'_{A} and c'_T, that is, $c'_T(r_0, r_{\pi'_{\mathrm{A}}(1)}) = \min_{r \in R} c'_T(r_0, r)$. The rest then follows using the same argument as in the proof of Lemma 3.8. According to the definition of the arrow protocol, $r_{\pi'_{\mathrm{A}}(1)}$ is the request corresponding to the first queuing message arriving at v_0. By definition of c'_T, this happens at time $c'_T(r_0, r_{\pi'_{\mathrm{A}}(1)})$. Because we assume all message delays to be at most 1, $c'_T(r_0, r)$ upper bounds the time when the message corresponding to r would reach v_0 for all other requests r. The lemma therefore follows. □

Combining inequality (12) with Lemma 3.20 and Theorem 3.18, we obtain the following theorem which extends Theorem 3.19 to the asynchronous setting.

Theorem 3.21. *Let* cost'_{arrow} *be the total cost of an asynchronous execution of the arrow protocol and let* cost'_{Opt} *be the total cost of an asynchronous execution of an optimal offline ordering algorithm. We have*

$$\rho' = \frac{\text{cost}'_{arrow}}{\text{cost}'_{Opt}} = O(s \cdot \log D),$$

where s *and* D *are the stretch and the diameter of the spanning tree* T*, respectively.*

4. Lower Bound

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

Theorem 4.1. *For any graph* G *and any spanning tree* T *of* G*, there is a set of ordering requests* R *such that the cost of the arrow protocol is a factor* $\Omega(s + \log D/\log\log D)$ *off the cost of an optimal ordering,* s *and* D *being the stretch and the diameter of the spanning tree* T*, respectively.*

Proof. We first prove the $\Omega(s)$ lower bound bound. By the definition of s, there are two nodes u and v for which $d_T(u, v) = s \cdot d_G(u, v)$. We place two requests r_u and r_v at the same time at nodes u and v, respectively. All algorithms (including the arrow protocol) need to send at least one message from u to v or from v to u. Because the arrow protocol communicates on the tree T, the queuing delay of arrow is at least $d_T(u, v)$ whereas an optimal algorithm can send the message with delay $d_G(u, v)$. Hence, the $\Omega(s)$ bound follows and it only remains to prove that the cost of the arrow protocol can be by a factor $\Omega(\log D/\log\log D)$ off the cost of an optimal ordering.

For the $\Omega(\log D/\log\log D)$ lower bound, we assume that the communication graph G is equal to T. Adding more edges to G can only decrease the cost of an optimal ordering. It is sufficient to concentrate on the nodes on a path P that induces 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 2 (if not, drop the part of P outside the largest possible power of 2).

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

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

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 9.

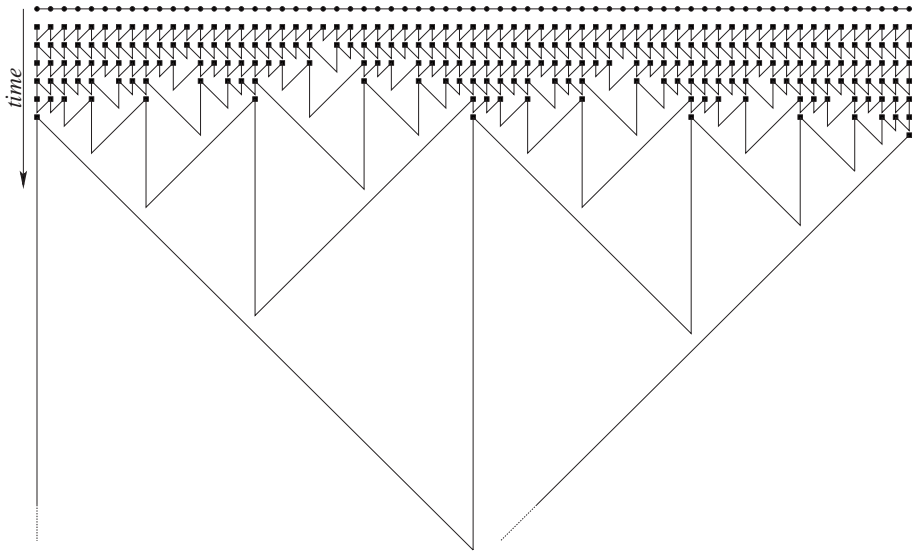

Fig. 9. A problem instance with diameter $D = 64$ and $k = 6$. The path is depicted horizontally, the time advances vertically. Each dot represents a request as computed by the recursion. The dots are connected by the arrow order π_A, starting with the root "virtual" request (top-left). The connection 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.

For this set of requests, from the definition of the recursion and as shown in Figure 9, 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 $\text{cost}_{arrow} = kD$.

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 $C_{\mathrm{M}}(MST) = O(D)$ for a sufficiently large D. Since an MST approximates an optimal order π_{O} within a factor of 2, and using the fact that cost_{Opt} is up to constants bounded from above by the Manhattan cost (see (3) and Definition 3.14), we conclude that $\mathrm{cost}_{Opt} = \mathrm{O}(D)$. Then the competitive ratio is

$$\rho = \frac{\mathrm{cost}_{arrow}}{\mathrm{cost}_{Opt}} = \frac{kD}{O(D)} = \Omega(k) = \Omega\left(\frac{\log D}{\log \log D}\right). \qquad \square$$

Note that for any stretch s, it is straightforward to construct a (G, T)-pair for which the competitive ratio of the arrow protocol is $O(s + \log D)$ which makes the above bound almost tight. One could for instance construct such a graph G by taking a tree and a cycle of length $s + 1$, connected by a single edge. However, as the next theorem shows, for every stretch s and every spanning tree T, there also is a graph G for which Theorem 3.19 is almost tight.

Theorem 4.2. *For any stretch s and any tree T, there is a graph G, such that s is the stretch of T and such that there is a set of ordering requests R such that the cost of the arrow protocol is a factor $\Omega(s \cdot \log(D/s)/\log\log(D/s))$ off the cost of an optimal ordering, D being the diameter of T.*

Proof. Let $P = v_0, \ldots, v_D$ be a path of length D of T, that is, P is a longest path of T. We construct a graph G by adding edges between nodes $v_{(i-1)s}$ and v_{is} for $i \in \{1, \ldots, \lfloor D/s \rfloor\}$. Clearly, the stretch of the spanning tree T of G is s. To place the requests, we need the set of requests for a path P' of length $\lfloor D/s \rfloor$ from the proof of Theorem 4.1. All requests which are assigned to the ith node of the path P' are placed on node $v_{(i-1)s}$. For the arrow protocol, the situation is exactly the same as in Theorem 4.1, except that each edge is replaced by a path of length s. The cost of the arrow protocol therefore is $\Omega(D \log(D/s)/\log\log(D/s))$. Because of the shortcuts from $v_{(i-1)s}$ to v_{is} on G, the cost of an optimal ordering is the same as in Theorem 4.1, that is, $O(D)$. Thus, the claim follows. $\square$

5. Experimental Results

Our theoretical analysis has so far shown that the arrow protocol is competitive to the optimal queuing algorithm under varying degrees of concurrency. We now present experimental results to show that in practice, the arrow protocol indeed performs very well, especially under situations of high concurrency.

We implemented and compared the arrow protocol and the centralized queuing protocol on an IBM SP2 distributed memory system with 76 processors. All programs were written using MPI (Message Passing Interface) [16], [8].

Arrow Protocol. Since the message latency between any pair of nodes in the SP2 machine was roughly the same, we could treat the network as a complete graph with all edges having the same weight. For the arrow protocol, we used a perfectly balanced binary tree ($\lfloor \log_2 n \rfloor$ depth for n nodes) as the spanning tree.

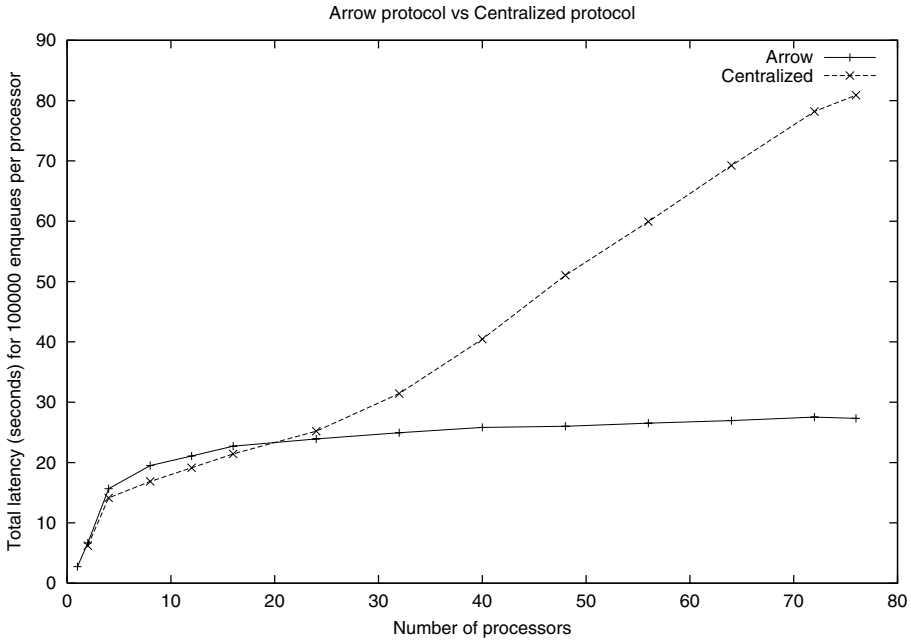

Fig. 10. Latency of the arrow protocol versus centralized protocol, for 100,000 enqueues per processor.

Centralized Protocol. A globally known central node always stored the current tail of the total order. Every queuing request was completed using only two messages, one to the central node, and one back.

We measured the time taken for 100,000 queuing requests per processor. Since our aim was to measure purely the synchronization cost, a processor's queuing request was considered complete when the request found its predecessor and the identity of the predecessor was returned to the processor. Each processor issued the next queuing request immediately after it learnt about the completion of its previous request.

The results of running this experiment on different sizes of the distributed system is shown in Figure 10. Figure 11 shows the average number of hops (interprocessor messages) to complete one request of the arrow protocol. The average number of hops is less than one because a large number of the requests find their predecessors locally, and thus generate zero interprocessor messages.

Discussion. Note that the total number of queuing requests issued by the system (100,000 per processor) increases linearly with the size of the system. The best we can hope for is that the latency remains constant with increasing system size, which would happen under conditions of ideal parallelism. No queuing algorithm can achieve this, since coordination between different processors is necessary to form a total order. In this light, it is surprising that the arrow protocol initially shows sub-linear slowdown and then remains nearly constant with increasing system size. In contrast, the centralized protocol shows a linear slowdown with increasing system size. This is to be expected, since the central

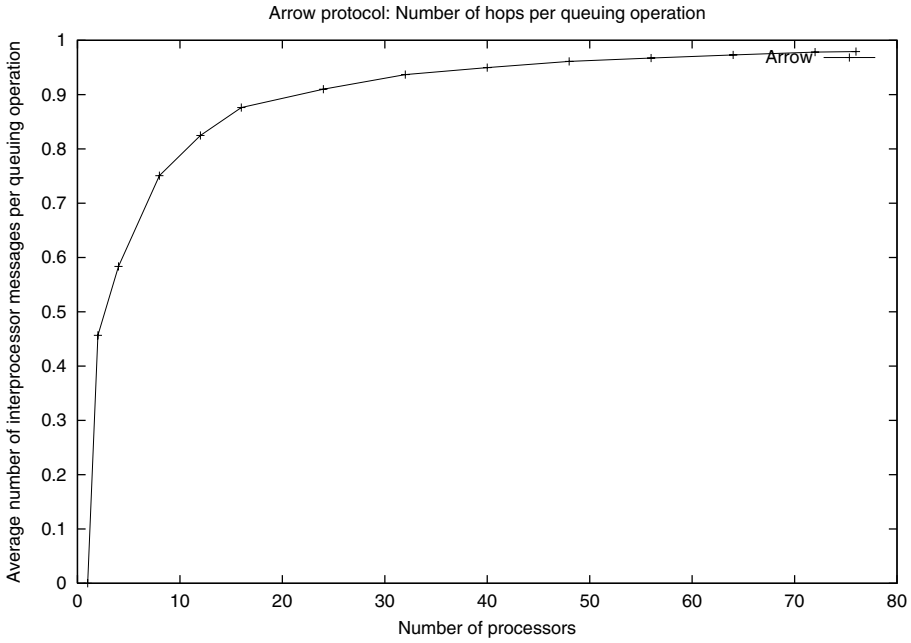

Fig. 11. Average number of hops per queuing request for the arrow protocol, taken over 100,000 requests per processor.

processor has to handle a linearly increasing number of messages with increasing system size. It is a tribute to the designers of the IBM SP2 that the system showed a graceful degradation (only a linear slowdown) under such loads. When we tried the same experiments on a loosely coupled network of SUN workstations, the centralized protocol could not scale beyond 20 processors, while the arrow protocol scaled easily.

These experiments suggest that the arrow protocol performs very well under concurrency. Queuing latencies are low and neighboring requests in the queue are very close on the tree.

5.1. *Related Experiments*

Another set of experiments on the arrow protocol were performed by Herlihy and Warres [12]. They used the protocol for building distributed directories, and compared it against a home-based (centralized) directory protocol. They observed that the arrow protocol outperformed the home-based protocol over a range of system sizes, from 2 to 16 processing elements.

Our experiments differ in the following aspects. Firstly, our experiments were conducted on a much larger scale system of up to 76 processors. Secondly, we were only interested in the queuing aspect of the protocol. Thus, we measured purely the queuing cost, while Herlihy and Warres [12] measured the total cost for maintaining the distributed directory, which included the additional cost of transferring the object down the queue.

References

[1] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proc*. 37*th IEEE Symposium on Foundations of Computer Science* (*FOCS*), pages 184–193, 1996.

[2] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proc*. 30*th Annual ACM Symposium on Theory of Computing* (*STOC*), pages 161–168, 1998.

[3] M. Charikar, C. Chekuri, A. Goel, S. Guha, and S. Plotkin. Approximating a finite metric by a small number of tree metrics. In *Proc*. 39*th IEEE Symposium on the Foundations of Computer Science* (*FOCS*), pages 379–388, 1998.

[4] M. Demmer and M. Herlihy. The Arrow Distributed Directory Protocol. In *Proc*. 12*th International Symposium on Distributed Computing* (*DISC*), pages 119–133, 1998.

[5] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[6] Y. Emek and D. Peleg. Approximating minimum max-stretch spanning trees on unweighted graphs. In *Proc*. 15*th ACM–SIAM Symposium on Discrete Algorithms* (*SODA*), pages 261–270, 2004.

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

[8] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI—The Complete Reference*, *Volumes* 1 *and* 2. Scientific and Engineering Computation Series. The MIT Press, Cambridge, MA, 2000.

[9] M. Herlihy and S. Tirthapura. Self-stabilizing distributed queuing. In *Proc*. 15*th International Symposium on Distributed Computing* (*DISC*), pages 209–223, 2001.

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

[11] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–96, 2001.

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

[13] T. Hu. Optimum communication spanning trees. *SIAM Journal on Computing*, 3(3):188–195, 1974.

[14] F. Kuhn and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. In *Proc*. 16*th ACM Symposium on Parallelism in Algorithms and Architectures* (*SPAA*), pages 294–301, 2004.

[15] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.

[16] MPI. The official MPI standard. http://www.mpi-forum.org.

[17] M. Naimi, M. Trehel, and A. Arnold. A $\log(n)$ distributed mutual exclusion algorithm based on path reversal. *Journal on Parallel and Distributed Computing*, 34(1):1–13, 1996.

[18] D. Peleg and E. Reshef. A variant of the Arrow Distributed Directory Protocol with low average complexity. In *Proc*. 26*th International Colloquium on Automata Languages and Programming* (*ICALP*), pages 615–624, 1999.

[19] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.

[20] 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.

Received October 18, 2004, *and in final form May* 5, 2006. *Online publication September* 14, 2006.