Dynamic Analysis of the Arrow Distributed Directory Protocol in General Networks*

Abdolhamid Ghodselahi[1] and Fabian Kuhn[2]

1 **Department of Computer Science, University of Freiburg, Germany**
`hghods@cs.uni-freiburg.de`
2 **Department of Computer Science, University of Freiburg, Germany**
`kuhn@cs.uni-freiburg.de`

Abstract

The Arrow protocol is a simple and elegant protocol to coordinate exclusive access to a shared object in a network. The protocol solves the underlying distributed queueing problem by using path reversal on a pre-computed spanning tree (or any other tree topology simulated on top of the given network).

It is known that the Arrow protocol solves the problem with a competitive ratio of $O(\log D)$ on trees of diameter D. This implies a distributed queueing algorithm with competitive ratio $O(s \log D)$ for general networks with a spanning tree of diameter D and stretch s. In this work we show that when running the Arrow protocol on top of the well-known probabilistic tree embedding of Fakcharoenphol, Rao, and Talwar [STOC 03], we obtain a randomized distributed online queueing algorithm with expected competitive ratio $O(\log n)$ against an oblivious adversary even on general n-node network topologies. The result holds even if the queueing requests occur in an arbitrarily dynamic and concurrent fashion and even if communication is asynchronous. The main technical result of the paper shows that the competitive ratio of the Arrow protocol is constant on a special family of tree topologies, known as hierarchically well separated trees.

1998 ACM Subject Classification F.1.2 Online computation, F.2.2 Computations on discrete structures, G.2.2 Network problems

Keywords and phrases Arrow protocol, competitive analysis, distributed queueing, shared objects, tree embeddings

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.22

1 Introduction

Coordinating the access to shared data is a fundamental task that is at the heart of almost any distributed system. For example, when implementing a distributed shared memory system on top of a message passing system, each shared register has to be kept in a coherent state despite possibly a large number of concurrent requests to read or write the shared register. In a distributed transactional memory system, each transaction might need to operate on several shared objects, which need to be kept in a consistent state [17, 26, 30]. When implementing a shared object on top of large-scale network, a *distributed directory protocol* can be used to improve scalability of the system [1, 2, 4, 6, 7, 17, 26]. When a network node requires access to a shared object, the directory moves a copy of the object

* A full version of this paper is available at `https://arxiv.org/abs/1705.07327` [11].

31st International Symposium on Distributed Computing (DISC 2017).
Editor: Andréa W. Richa; Article No. 22; pp. 22:1–22:16

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to the node requesting the object. If the node changes the state of the shared object, the directory protocol has to make sure that all existing copies of the object are kept in a consistent state.

Distributed Queueing. At the core of many distributed directory implementations is the following basic *distributed queueing problem* that allows the system to order potential concurrent access requests to a shared object [15]. The nodes of a network issue queueing requests (e.g., requests to access a shared object) in a completely dynamic and possibly arbitrarily concurrent manner. A queueing protocol needs to globally order all the requests so that they can be acted on consecutively. Formally, each request has to find its *predecessor request* in the order. That is, when enqueueing a request r issued by some node v, a queueing protocol needs to find the request r' that currently forms the tail of the queue and inform the node v' of request r' about the new request r. The cost of enqueueing request r after request r' is defined as the delay from when request r is issued until when the node v' knows that r is the successor request of r'.

The Arrow Protocol. A particularly simple and elegant solution for this distributed queueing problem is given by the Arrow protocol, which was introduced independently (in slightly different forms) by Naimi and Trehel, Raymond, as well as van de Snepscheut in the context of distributed mutual exclusion [21, 24, 29]. The Arrow protocol operates on a directed tree topology $T = (V, E)$. In a quiescent state, the tree is rooted at the node u of the current tail of the queue, i.e., all edges of T are directed towards u. When a new queueing request is issued at a node v, the directions of the edges on the path between v and the previous tail u are reversed so that the tree is now rooted at v. For a precise description of the protocol, we refer to Section 2. It has been shown in [8] that the Arrow protocol correctly solves the queueing problem even in an asynchronous system even if the requests are issued in a completely dynamic and possibly concurrent way. Moreover, the Arrow protocol guarantees that every request finds the node of its predecessor on a direct path (i.e., within D time units if D is the diameter of T). The overall cost of some queueing algorithm is the sum of delays between when each request is issued and when its predecessor in the final order knows about it. In [14], it was further shown that on a tree T, the overall cost of the Arrow protocol for ordering a dynamic set of queueing requests is within a factor $O(\log D)$ of the cost of an optimal offline queueing algorithm, which knows the request sequence in advance.[1]

Contribution. In the present paper, we strengthen the result of [14] and we show that when run on the right underlying tree, the Arrow protocol is $O(\log n)$-competitive even on general n-node network topologies. The competitive ratio achieved by the Arrow protocol is the worst case ratio between the cost of Arrow and the cost of an optimal offline ordering strategy. The best previously known competitive ratio for the distributed queueing problem with arbitrarily dynamically injected requests on general graphs is $O(\log^2 n \cdot \log D)$ as shown in [27] for the hierarchical schemes defined of [4, 26]. This shows that (under some assumptions), the simple and elegant Arrow protocol outperforms all existing significantly more complicated distributed queueing protocols. We note that our protocol is based on a randomized tree construction and its competitive ratio is w.r.t. an oblivious adversary (i.e., the adversary needs to determine the sequence of requests before the construction of the tree). Other

[1] Note that this implies a competitive ratio of $O(s \cdot \log D)$ for general graphs if a spanning tree T of diameter D and stretch s is given.

protocols with polylogarithmic competitive ratio are deterministic and they therefore also work in the presence of an adaptive adversary. For a more detailed comparison of our results with existing protocols, we refer to the discussion in Section 1.1.

More specifically, as our main technical result, we show that the Arrow protocol is $O(1)$-competitive when it is run on a special class of trees known as *hierarchically well separated trees* [5]. A hierarchically well separated tree (in the following referred to as an HST) with parameter $\alpha > 1$ is a weighted rooted tree, where on each level, all the nodes are at the same distance to the root (the distance to the root depend on α and on the level) and all the leaves are on the same level (and thus also at the same distance to the root). Further, the edge lengths decrease exponentially (by a factor α per level) when going from the root towards the leaves. The properties of HSTs as well as the way we utilize HSTs are formally described in Section 2. When running Arrow on an HST T, we assume that all requests are issued at the leaves of T. We show that the total cost of an Arrow execution on an HST T is within a constant factor of the total cost of an optimal offline algorithm for the given set of requests. Our result even holds if the communication on T is asynchronous.

▶ **Theorem 1.** *Assume that we are given an HST T with parameter $\alpha = 2$ and queueing requests R that arrive in an arbitrarily dynamic manner at the leaves of T. When using the* Arrow *protocol on tree T, the total cost for ordering the requests in R is within a constant factor of the cost of an optimal offline algorithm for ordering the requests R on T. This even holds if communication is asynchronous.*

▶ Remark. Because the statement of the theorem applies to the general asynchronous case, it also captures a synchronous scenario, where the delay on each edge is fixed, but might be smaller than the actual weight of the edge in the HST. Such executions are relevant because an HST T is often built as an overlay graph on top of an underlying network graph G, where each edge of weight w in T is mapped to a path of length at most w in G and thus even in a synchronous execution, the delay when sending a message across the edge of T might be smaller than the weight of the edge.

For a precise description of the Arrow protocol and the definition of queueing cost, we refer to Section 2. When combining Theorem 1 with the celebrated probabilistic tree embedding of Fakcharoenphol, Rao, and Talwar [9], we get our main result for general graphs. In [9], it is shown that there is a randomized algorithm that given an arbitrary n-point metric (X, d) constructs an HST T such all points X are mapped to leaves of T, all distances in (X, d) are upper bounded by the respective distances in T, and the expected distance between any two leaves in T is within an $O(\log n)$ factor of the distance between the corresponding two points in X. When constructing such an HST T for a given graph G and when assuming an oblivious adversary[2], this implies that the expected total cost of Arrow on T is within an $O(\log n)$ factor of the optimal offline queueing cost on G. We also note that an efficient distributed construction of the HST embedding of [9] has been given in [10].

▶ **Theorem 2.** *Assume that we are given an arbitrary graph $G = (V, E)$ and queueing requests R that arrive in an arbitrarily dynamic manner at the nodes of G. There is a randomized construction of an HST T that can be simulated on G such that when running* Arrow *on T, we get a distributed queueing algorithm for G with expected competitive ratio at most $O(\log n)$ against an oblivious adversary providing the sequence of requests. This even holds if communication is asynchronous.*

[2] That is, when assuming that the sequence of requests is statistically independent of the randomness used to construct the HST T or equivalently, if the adversary determines the sequence of requests before the tree T is constructed.

Organization of the Paper. The remainder of the paper is organized as follows. Section 2 formally defines the queueing problem, the Arrow protocol, as well as the cost model used in our paper. The section also contains some lemmas that establish some basic properties that are needed for the rest of the paper. Section 3 analyzes the cost of an optimal offline algorithm on an HST T by relating it to the total weight of an MST defined on the set of requests. In Section 4, we introduce a general framework to analyze the queueing cost of distributed queueing algorithms on an HST T and the framework is applied to synchronous executions of the Arrow protocol. For the analysis of asynchronous executions, we refer to the full version of the paper [11]. Due to lack of space, we also need to omit most proofs throughout the technical sections of the paper. All the missing proofs can also be found in the full version [11].

1.1 Related Work

The Arrow protocol has been introduced (in somewhat different forms) by Naimi and Trehel, Raymond, as well as van de Snepscheut [24] as a way to solve the mutual exclusion problem in a network. The protocol was later reinvented by Demmer and Herlihy [8], who used Arrow to implement a distributed directory [6]. Over the years, Arrow has been used and analyzed in different contexts [14, 16, 18, 19, 22, 28]. The protocol has been implemented as a part of Aleph Toolkit [16] and shown to outperform centralized schemes significantly in practice [19]. Several other tree-based distributed queueing protocols that are similar to the Arrow protocol have also been proposed in the literature. A protocol that combines the ideas of Arrow with path compression has been implemented in the Ivy system [20]. The amortized cost to serve a single request is only $O(\log n)$ [12], however the protocol needs a complete graph as the underlying network topology. There are also other similar protocols that operate on fixed trees. The Relay protocol [30] has been introduced as a distributed transactional memory protocol. It is run on top of a fixed spanning tree similar to Arrow, however to more efficiently deal with aborted transactions, it does not always move the shared object to the node requesting it. Further, in [2], a distributed directory protocol called Combine has been proposed. Combine runs on a fixed overlay tree and it is in particular shown in [2] that Combine is starvation-free.

The first paper to study the competitive ratio of concurrent executions of a distributed queueing protocol is [15]. The paper shows that in synchronous executions of Arrow on a tree T, if all requests are issued at time 0 (known as one-shot executions), the total cost of Arrow is within a factor $O(\log |R|)$ compared with the optimal queueing cost on tree T. This analysis has later been extended (and slightly strengthened) to the general concurrent setting where requests are issued in an arbitrarily dynamic fashion. In [14], it is shown that in this case, the total cost of Arrow is within a factor $O(\log D)$ of the optimal cost on the tree T. Later, the same bounds have also been proven for the Relay protocol [30] and the Combine protocol [2]. Typically, these protocols are run on a spanning tree or an overlay tree on top of an underlying general network topology. While the cost of all these protocols is small when compared with the optimal queueing cost on the tree, the cost of the protocols might be much larger when compared with the optimal cost on the underlying topology. In this case, the competitive ratio becomes $O(s \cdot \log D)$, where s is the stretch of the tree. There are underlying graphs (e.g., cycles) for which every spanning tree and even every overlay tree has stretch $\Omega(n)$ [13, 23]. The fact that even the best spanning tree might have large stretch initiated the work on distributed queueing protocols that run on more general hierarchical structures. In [17], a protocol called Ballistic is introduced and analyzed for the sequential and the one-shot case. Ballistic has competitive ratio $O(\log D)$, however the protocol requires

the underlying distance metric to have bounded doubling dimension and it thus cannot be applied in general networks. The best protocol known for general networks is Spiral, which was introduced in [26]. Spiral is based on a hierarchy of overlapping clusters that cover the graph. It's general structure is thus somewhat resembling the classic sparse partitions and mobile objects solutions by Awerbuch and Peleg [3, 4]. The competitive ratio of Spiral is shown to be $O(\log^2 n \cdot \log D)$ for sequential and one-shot executions in [26]. In [27], a general framework to analyze the cost of concurrent executions of hierarchical queueing and directory protocols has been presented. In particular, in [27], the competitive analysis of Spiral and also of the classic mobile object algorithm of Awerbuch and Peleg [3, 4] has been extended to the dynamic setting. In [14], it is sketched how the competitive analysis for Arrow generalizes to the asynchronous case.

2 Model, Problem Statement, and Preliminaries

Communication Model. We consider a standard message passing model on a network modeled by a graph $G = (V, E)$. In some cases, the edges of G have weights $w : E \to \mathbb{R}_{>0}$, which are assumed to be normalized such that $w(e) \geq 1$ for all $e \in E$. We distinguish between synchronous and asynchronous executions. In a *synchronous execution*, the delay for sending a message from a node u to a node v over an edge e connecting u and v is exactly 1 if the edge is unweighted and exactly $w(e)$ otherwise. In an *asynchronous execution*, message delays are arbitrary, however when analyzing an asynchronous execution, we assume that the message delay over an edge e is upper bounded by the edge weight $w(e)$ (or by 1 in the unweighted case).

The Distributed Queueing Problem. In the *distributed queueing problem* on a graph $G = (V, E)$, a set R of queueing requests $r_i = (v_i, t_i)$ are issued at the nodes of V (every node can issue multiple requests) in an arbitrarily dynamic fashion. The goal of a queueing algorithm is to order all the requests. Specifically, if a request $r_i = (v_i, t_i)$ is issued at node v_i at time $t_i \geq 0$, the algorithm needs to enqueue the request r_i by informing the node v_j of the predecessor request $r_j = (v_j, t_j)$ in the constructed global order. For this purpose, every queueing algorithm in particular has to send (possibly indirectly) a respective message from node v_i to v_j. We assume that at time 0, when an execution starts, the tail of the queue is at a given node $v_0 \in V$. Formally, this is modeled as a request $r_0 = (v_0, 0)$ which has to be ordered first by any queueing protocol. We sometimes refer to r_0 as the dummy request. For a set R' of queueing requests (and sometimes by overloading notation also for a set of request indexes), we define $t_{\min}(R')$ and $t_{\max}(R')$ to be the minimum and the maximum issue time t of any request $r = (v, t) \in R'$, respectively.

The Arrow Protocol. The Arrow protocol [24] is a distributed queueing protocol that operates on a tree network $T = (V, E)$. At each point in time, each node $v \in V$ has exactly one outgoing link (arrow) pointing either to one of the neighbors of v or to the node v itself. In a quiescent state, the arrow of the node of the request at the tail of the queue points to itself and all other arrows point towards the neighbor on the path towards the tail of the queue (i.e., the tree is directed towards the current tail). When a new request at a node $v \in V$ occurs, a "find predecessor" message is sent along the arrows until it finds the predecessor request. While following the path to the direction of the arrows are reversed. More formally, a request r at node v is handled as follows.

1. If the arrow of v points to v itself, r is queued directly behind the previous request issued at v. Otherwise if the arrow points to neighbor u, *atomically*, a "find predecessor" message (including the information about request r) is sent to u and the arrow of v is redirected to v itself.
2. If a node u receives a "find predecessor" message for request r from a neighbor w, if the arrow of u points to itself, *atomically*, the request r is queued directly behind the last request issued by node u and the arrow of u is redirected to node w. Otherwise, if the arrow of u points to neighbor x, *atomically*, the "find predecessor" message is forwarded to node x and the arrow of node u is redirected to node w.

For a more detailed description of the Arrow protocol and of how Arrow handles concurrent requests, we refer the reader to [8, 14]. It was shown in [8] that the Arrow protocol correctly orders a given sequence of requests even in an asynchronous network. Moreover as shown in [8, 14], when operating on tree T, the protocol always finds the predecessor of a request on the direct path on T. As a result, if two requests r' and r are at distance d on T and if r' is the predecessor of r in the queueing order, the "find predecessor" message initiated by request r finds the node of request r' in time exactly d in the synchronous setting and in time at most d in the asynchronous model. Further, it is shown in [14] that the successor request of a request r at node v in the queue is always the remaining request r'' that first reaches v on a direct path. This "greedy" nature of the Arrow ordering was used in [15], where it was shown that in the one-shot case when all requests occur at time 0, the Arrow order corresponds to a greedy (nearest neighbor) TSP path through requests, whereas an optimal offline algorithm corresponds to an optimal TSP path on the request set. The competitive ratio on trees then follows from the fact that the nearest neighbor heuristic provides a logarithmic approximation of the TSP problem [25]. In [14], this analysis was extended and it was shown that even in the fully dynamic case, it is possible to reduce the problem to a (generalized) TSP nearest neighbor analysis. Formally, the greedy nature of the Arrow protocol in the synchronous setting is captured by Lemma 7 in Section 3.

Hierarchically Well Separated Trees. The notion of a *hierarchically well separated tree* (HST) was defined by Bartal in [5]. Given a parameter $\alpha > 1$, an HST of depth h is a rooted tree with the following properties. All children of the root are at distance α^{h-1} from the root. Further, every subtree of the root is an HST of depth $h-1$ that is characterized by the same parameter α (i.e., the children 2 hops away from the root are at distance α^{h-2} from their parents). The probabilistic tree embedding result of [9] shows that for every metric space (X, d) with minimum distance normalized to 1 and for every constant $\alpha > 1$, there is a randomized construction of an HST T with a bijection f of the points in X to the leaves of T such that for every $x, y \in X$, $d(x, y) \leq d_T(f(x), f(y))$ and such that the expected tree distance $\mathbb{E}\big[d_T(f(x), f(y))\big] = O(\log |X|) \cdot d(x, y)$. Further, an efficient distributed implementation of the construction of [9] for the distances of a given network graph was given in [10].

The main technical result of this paper is an analysis of Arrow on an HST T if all requests are issued at leaves of T. Throughout the paper, the HST parameter α is set to $\alpha = 2$. For convenience, we number the levels of an HST T of depth h from 0 to h, where the level 0 nodes are the leaves and the single level h node is the root. For $\ell \in \{0, \ldots, h\}$, $\delta(\ell) := 2^{\ell+1} - 2$ denotes the distance between two leaves for which the least common ancestor is on level ℓ.

Cost Model. Assume when applying some queueing algorithm ALG to the dynamic set of requests R, the requests are ordered according to the permutation π_{ALG} such that the request ordered at position i in the order is $r_{\pi_{\mathsf{ALG}}(i)}$. For every $i \in \{1, \ldots, |R| - 1\}$, we define

the *cost of ordering* $r_{\pi_{\mathsf{ALG}}(i)}$ *after* $r_{\pi_{\mathsf{ALG}}(i-1)}$ as the time it takes the queueing algorithm ALG to enqueue the request $r_{\pi_{\mathsf{ALG}}(i)}$ as the successor of $r_{\pi_{\mathsf{ALG}}(i-1)}$. More specifically, we assume that request $r_{\pi_{\mathsf{ALG}}(i)}$ can be enqueued as soon as the predecessor request $r_{\pi_{\mathsf{ALG}}(i-1)}$ is in the system and as soon as node $v_{\pi_{\mathsf{ALG}}(i-1)}$ knows about request $r_{\pi_{\mathsf{ALG}}(i)}$. Assume that algorithm ALG informs node $v_{\pi_{\mathsf{ALG}}(i-1)}$ (through a message) about $r_{\pi_{\mathsf{ALG}}(i)}$ at time $t_{\mathsf{ALG}}(i)$. The cost (latency) $L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i-1)}, r_{\pi_{\mathsf{ALG}}(i)})$ incurred for enqueueing request $r_{\pi_{\mathsf{ALG}}(i)}$ and the overall cost (latency) $cost_{\mathsf{ALG}}$ of ALG are then defined as follows.

$$L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i-1)}, r_{\pi_{\mathsf{ALG}}(i)}) := \max\left\{t_{\mathsf{ALG}}(i), t_{\pi_{\mathsf{ALG}}(i-1)}\right\} - t_{\pi_{\mathsf{ALG}}(i)}, \tag{1}$$

$$cost_{\mathsf{ALG}}(\pi_{\mathsf{ALG}}) := \sum_{i=1}^{|R|-1} L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i-1)}, r_{\pi_{\mathsf{ALG}}(i)}). \tag{2}$$

We next specify the above cost more concretely for Arrow and for an optimal offline algorithm. Assume that we have an execution $\mathcal{A}$ of the Arrow protocol that operates on a tree T. Let $\pi_{\mathcal{A}}$ be the ordering induced by the Arrow execution $\mathcal{A}$. When the "find predecessor" message of a request $r_{\pi_{\mathcal{A}}(i)}$ arrives at the node of the predecessor request $r_{\pi_{\mathcal{A}}(i-1)}$, clearly the request $r_{\pi_{\mathcal{A}}(i-1)}$ has already occurred and thus we always have $L_{\mathcal{A}}(r_{\pi_{\mathcal{A}}(i-1)}, r_{\pi_{\mathcal{A}}(i)}) = t_{\mathcal{A}}(i) - t_{\pi_{\mathcal{A}}(i)}$ for any Arrow execution. Further note, that in a synchronous execution of arrow on tree T, because Arrow always finds the predecessor on the direct path, this latency cost is always equal to the distance between the respective nodes in T.

When studying the cost of an optimal offline queueing algorithm $\mathcal{O}$, we assume that $\mathcal{O}$ knows the whole sequence of requests in advance. However, $\mathcal{O}$ still needs to send messages from each request to its predecessor request. The message delays are not under the control of the optimal offline algorithm. When lower bounding the cost of $\mathcal{O}$, we can therefore assume that all communication is synchronous even in the asynchronous case. Note that a synchronous execution is a possible strategy of the asynchronous scheduler. When operating on a graph G, the latency cost of $\mathcal{O}$ for ordering a request r_j as the successor of a request r_i is then exactly $L^G_{\mathcal{O}}(r_i, r_j) = \max\{t_i - t_j, d_G(v_i, v_j)\}$. As we analyze Arrow on an HST T that is simulated on top of an underlying network G, we directly define the optimal offline w.r.t. synchronous executions on the tree T as follows.

$$L^T_{\mathcal{O}}(r_{\pi^T_{\mathcal{O}}(i-1)}, r_{\pi^T_{\mathcal{O}}(i)}) := \max\left\{d_T(v_{\pi^T_{\mathcal{O}}(i-1)}, v_{\pi^T_{\mathcal{O}}(i)}), t_{\pi^T_{\mathcal{O}}(i-1)} - t_{\pi^T_{\mathcal{O}}(i)}\right\}, \tag{3}$$

$$cost^T_{\mathcal{O}}(\pi^T_{\mathcal{O}}) := \sum_{i=1}^{|R|-1} L^T_{\mathcal{O}}(r_{\pi^T_{\mathcal{O}}(i-1)}, r_{\pi^T_{\mathcal{O}}(i)}). \tag{4}$$

The ordering $\pi_{\mathcal{O}}$ is chosen such that the total cost $cost^T_{\mathcal{O}}(\pi_{\mathcal{O}})$ in (4) is minimized. The next lemma shows that when using the randomized HST construction of [9], the cost (4) is within a logarithmic factor of the optimal offline cost on the underlying network graph G.

▶ **Lemma 3.** *Assume T is an HST that is constructed on top of an n-node network graph G by using the randomized algorithm of [9] and assume that there is a dynamic set of queueing requests issued at the nodes of G. If the sequence of requests is independent of the randomness of the randomized HST construction, the expected optimal total cost on T (as defined in (4)) is within a factor $O(\log n)$ of the optimal offline queueing cost on G.*

Given Theorem 1 (which will be proven as the main technical result of the paper) and Lemma 3, we immediately get Theorem 2. We note in light of the remark following the statement of Theorem 1 in Section 1, the statement of Theorem 2 is also true for synchronous executions on the underlying graph G.

Manhattan Cost. In the dynamic competitive analysis of Arrow on general trees in [14], it has been shown that it is useful to study the optimal ordering w.r.t. to the following *Manhattan cost* on a tree T between two queueing requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$.

$$c_{\mathcal{M}}^T(r_i, r_j) := d_T(v_i, v_j) + |t_i - t_j|. \tag{5}$$

As the cost function $c_{\mathcal{M}}(r_i, r_j)$ defines a metric space on the request set, the problem of finding an optimal ordering w.r.t. the cost $c_{\mathcal{M}}(r_i, r_j)$ is a metric TSP problem.[3] As a result, we will for example use that the total weight of an MST on the set of request w.r.t. the weight function $c_{\mathcal{M}}(r_i, r_j)$ is within a factor 2 of the cost of an optimal TSP path. The following definition is inspired by Lemma 3.12 in [14].

▶ **Definition 4** (Condensed Request Set)**.** A set R of queueing requests $r_i = (v_i, t_i)$ on a tree T is called *condensed* if for any two requests $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ that are consecutive w.r.t. time of occurrence, there exists requests $r_a = (v_a, t_a)$ and $r_b = (v_b, t_b)$ such that $t_a \leq t_i$, $t_b \geq t_j$, and $d_T(v_a, v_b) \geq t_b - t_a$.

It is shown in [14] that for condensed request sets, the total optimal Manhattan cost is within a constant factor of the optimal offline queueing cost.

▶ **Lemma 5** (Lemma 3.17 in [14] rephrased)**.** *If the request set R is condensed, then on any tree T and for every ordering π on the requests, it holds that*

$$\sum_{i=1}^{|R|-1} c_{\mathcal{M}}^T(r_{\pi(i-1)}, r_{\pi(i)}) \leq 12 \cdot \sum_{i=1}^{|R|-1} L_{\mathcal{O}}^T(r_{\pi(i-1)}, r_{\pi(i)}).$$

For synchronous executions on trees, it is also shown in [14] that every request set R can be transformed into a condensed request set without changing the ordering (and the cost) of Arrow and without increasing the optimal offline cost.

▶ **Lemma 6** (Lemma 3.11 in [14] rephrased)**.** *Let R be a set of queueing requests issued on a tree T and let $r_i = (v_i, t_i)$ and $r_j = (v_j, t_j)$ be two requests of R that are consecutive w.r.t. time of occurrence. Further, choose two requests $r_a = (v_a, t_a)$ with $t_a \leq t_i$ and $r_b = (v_b, t_b)$ with $t_b \geq t_j$ minimizing $\delta := t_b - t_a - d_T(v_a, v_b)$. If $\delta > 0$, every request $r = (v, t)$ with $t \geq t_j$ can be replaced by a request $r' = (v, t - \delta)$ without changing the synchronous* Arrow *order and without increasing the optimal offline cost.*

Lemma 6 implies that every request set R can be transformed into a condensed set R' without changing the synchronous order of Arrow and without increasing the optimal offline cost. For the analysis of Arrow in synchronous systems, we can thus w.l.o.g. assume that the request set is condensed.

3 Analysis of the Optimal Offline Cost

This and the next section discuss the main technical contribution of the paper and analyzes the total cost of a synchronous Arrow execution when run on an HST T. Throughout this section, we assume that a fixed HST T, a set of dynamic requests R placed at the leaves of T, and a synchronous execution of Arrow with request set R on T are given. For convenience, we

[3] The relation of Arrow and the TSP problem was already exploited in [14] when analyzing Arrow on general trees.

relabel the requests in R so that they are ordered according to the queueing order resulting from the given Arrow execution on T. That is, we assume that for all $i \in \{0, \ldots, |R|-1\}$, request $r_i = (v_i, t_i)$ is the i^{th} request in Arrow's order. Note that $r_0 = (v_0, 0)$ is still the dummy request defining the initial tail of the queue. As discussed in Section 2, the Arrow order can be seen as a greedy ordering in the following sense. Given the first $i-1$ requests in the order, the i^{th} request r_i is a request $r = (v, t)$ from the subset of the remaining requests that can reach the node v_{i-1} of request r_{i-1} first immediately sending a message at time t from node v to node v_{i-1}. This greedy behavior is captured by the following basic lemma. For a more thorough discussion, we refer to [14].

▶ **Lemma 7.** *Consider a synchronous execution of* Arrow *on tree T and consider two arbitrary requests r_i and r_j for which $1 \leq i < j$ (i.e., r_j is ordered after r_i by* Arrow*). Then it holds that*

1. $t_i + d_T(v_{i-1}, v_i) \leq t_j + d_T(v_{i-1}, v_j)$ *and*
2. $t_i \leq t_j + d_T(v_i, v_j)$.

Before delving into the details of the analysis, we give a short outline. In the first step in Section 3.1, we study the ordering generated by Arrow in more detail and show that it implies a hierarchical partition of the requests R in a natural way. To simplify the next step, Section 3.2 transforms the given HST T into a new tree such that inside each subtree, if ordering the request by time of occurrence, the gap between the times of consecutive requests cannot be too large (whenever such a gap is too large, we split the corresponding subtree into two trees). Section 3.3 then shows that the optimal offline cost can be characterized by the total Manhattan cost of a spanning tree that respects the hierarchical structure of the HST T in a best given way. Finally, in Section 4, we give a general framework to compare the queueing cost of an online distributed algorithm on an HST T to the optimal offline cost on T and we apply this method to synchronous Arrow executions.

3.1 Characterizing Arrow By A Hierarchical Partition of R

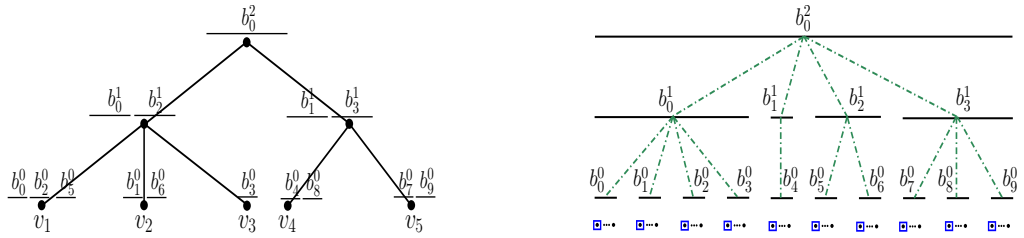
We hierarchically partition the requests R according to the Arrow queueing order and the hierarchical structure of the HST T. On each level ℓ of T, we partition the requests into blocks, where a block of requests is a maximal set of requests that are ordered consecutively by Arrow inside some level-ℓ subtree of T. In the following, for non-negative integers s and t, we use the abbreviations $[s] := \{0, \ldots, s-1\}$ and $[s, t] := \{s, \ldots, t\}$. Formally, instead of partitioning the set of requests R directly, we partition the set of indexes $[|R|]$. Recall that the requests in R are indexed consecutively according to the queueing order of Arrow.

▶ **Definition 8** (Hierarchical Block Partition)**.** For each level $\ell \in [0, h]$, we partition $[|R|]$ into $n(\ell)$ blocks $\left\{b_0^\ell, b_1^\ell, \cdots, b_{n(\ell)-1}^\ell\right\}$ such that

1. each block is a consecutive set of integers (i.e., a consecutively ordered set of requests),
2. for every block b_i^ℓ, all requests r_p for $p \in b_i^\ell$ are in the same level-ℓ subtree of T, and
3. for all $i, j \in [n(\ell)]$ and all $p \in b_i^\ell$ and $q \in b_j^\ell$, $i < j \implies p < q$.

For each block b, we define the first request of b to be the one with min. index in b.

Note that for each level ℓ and for the first block of this level, the first request of the block has index 0. The block partition defined in Definition 8 is illustrated in Figure 1. Figure 1a shows the blocks within the HST structure, whereas Figure 1b shows the hierarchical partition induced by the blocks. To simplify the presentation of our analysis, we also define a level -1 block b_i^{-1} for each individual request r_i. Note that we have $n(-1) = |R|$. The following definition allows us to navigate through the block hierarchy.



(a) Blocks within the same subtree. **(b)** Tree induced by the block hierarchy.

Figure 1 The partition of R. (a) An HST with height 2 and 5 leaves. The leaves issue requests at different times. The issued requests by nodes v_1, v_2, and v_3 are partitioned into the blocks b_0^1 and b_2^1 on level 1. These two blocks are called neighbor blocks at a subtree rooted at height 1. (b) The corresponding 4 level-wise partition based on Arrow's order that forms a parent-child relation between the blocks on different levels. Blue boxes include the requests that are ordered first by Arrow among all requests in blocks b_i^0 for all $i \in [0, 9]$.

▶ **Definition 9** (Children Blocks)**.** The set of children blocks of a block b_i^ℓ on a level $\ell \in [0, h]$ is defined as $child(b_i^\ell) := \left\{b_j^{\ell-1} : b_j^{\ell-1} \subseteq b_i^\ell\right\}$. Block b_i^ℓ is called the parent block of each of the blocks in $child(b_i^\ell)$.

In Figure 1b, block b_2^1 is the parent block of its children blocks b_5^0 and b_6^0. Block b_1^1 has only one child block b_4^0 and thus $b_1^1 = b_4^0$.

The blocks $\left\{b_0^\ell, b_1^\ell, \cdots, b_{n(\ell)-1}^\ell\right\}$ of level ℓ belong to the subtrees rooted at height ℓ of the HST T. Note that by the definition of the block partition, no two consecutive blocks at the same level ℓ belong to the same level-ℓ subtree of T. The next definition specifies notation to argue about blocks of the same subtree of T.

▶ **Definition 10** (Blocks of Same Subtree)**.** If two blocks b_i^ℓ and b_j^ℓ belong to the same level-ℓ subtree of T, this is denoted by $\widehat{b_i^\ell b_j^\ell}$. Moreover, $|\widehat{b_i^\ell b_j^\ell}| := \left|\left\{w : i < w < j \ \wedge\ \widehat{b_i^\ell b_w^\ell} \text{ holds}\right\}\right|$. Two blocks b_i^ℓ and b_j^ℓ are called *neighbor blocks* if $\widehat{b_i^\ell b_j^\ell}$ and $|\widehat{b_i^\ell b_j^\ell}| = 0$.

In Figure 1a, blocks b_0^0, b_2^0, and b_5^0 are within the same subtree rooted at node v_1. Blocks b_0^0 and b_5^0 are not neighbor blocks, however blocks b_0^0 and b_2^0, as well as blocks b_2^0 and b_5^0 are neighbor blocks. The next lemma lists a number of simple properties of the block partition.

▶ **Lemma 11.** *The hierarchical block partition of Def. 8 satisfies the following properties:*

1. *For every block b_i^ℓ and for all $p, q \in b_i^\ell$, we have $d_T(v_p, v_q) \leq \delta(\ell)$.*
2. *For each level ℓ and all level-ℓ blocks b_i^ℓ and b_j^ℓ, if $\widehat{b_i^\ell b_j^\ell}$ holds, for any $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) \leq \delta(\ell)$.*
3. *For each level ℓ and all level-ℓ blocks b_i^ℓ and b_j^ℓ, if $\widehat{b_i^\ell b_j^\ell}$ does* not *hold, for all $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) \geq \delta(\ell + 1)$.*
4. *Assume $\ell < h$ and consider two blocks b_i^ℓ and b_j^ℓ that have a common parent block $b_w^{\ell+1}$, but for which $\widehat{b_i^\ell b_j^\ell}$ does not hold. Then, for all $p \in b_i^\ell$ and $q \in b_j^\ell$, we have $d_T(v_p, v_q) = \delta(\ell+1)$.*

We have seen that in a synchronous Arrow execution, the latency cost for ordering request r_{i+1} as the successor of r_i is exactly the distance $d_T(v_i, v_{i+1})$ between the nodes of the two requests. The total cost of Arrow therefore directly follows from the structure of the block partition.

▶ **Lemma 12.** *The total cost of a synchronous* Arrow *execution on the HST T with corresponding hierarchical block partition is given by*

$$cost_{\mathcal{A}}(\pi_{\mathcal{A}}) = \sum_{\ell=0}^{h-1} \big(n(\ell) - n(\ell+1)\big) \cdot \delta(\ell+1).$$

3.2 HST Conversion

We next provide a recursive (top-down) splitting procedure that converts the original HST into a new HST with better properties. The conversion does not change the total cost of ordering the requests by Arrow (in fact, it does not change the block partition). Further, the total Manhattan cost of optimal offline algorithm's order asymptotically remains unchanged as well. We describe how the splitting procedure works and we then argue its properties.

Splitting Procedure. We describe the splitting procedure as it is applied to a subtree T' that is rooted at a given level $\ell \in \{0, \dots, h\}$ of T. If $\ell = 0$, the tree T' is returned unchanged. Otherwise ($\ell \geq 1$), we go through all level-$(\ell-1)$ subtrees T'' of T'. As long as the tree T'' has two neighbor blocks $b_i^{\ell-1}$ and $b_j^{\ell-1}$ (for $i < j$) for which the following condition (6) is true, the subtree T'' is split into two separate subtrees T_1'' and T_2'' of T'.

$$t_{\min}(b_j^{\ell-1}) - t_{\max}(b_i^{\ell-1}) \geq \delta(\ell). \tag{6}$$

The splitting of T'' into T_1'' and T_2'' works as follows. The topology of T_1'' and T_2'' is identical to the topology of T''. Each request $r = (v, t)$ that is issued at some node v of T'' is either placed on the isomorphic copy of v in T_1'' or in T_2''. All requests r in blocks $b_x^{\ell-1}$ of T'' for $x \leq i$ are placed in tree T_1'' and all request in blocks $b_y^{\ell-1}$ of T'' for $y \geq j$ are placed in tree T_2''. We perform such splittings for trees T' of level ℓ as long as there are subtrees of T' on level $\ell - 1$ with neighbor blocks that satisfy Condition (6). As soon as no such neighbor blocks exist, the procedure is applied recursively to all trees T'' at level $\ell - 1$ (incl. the new subtrees). The conversion is started by applying the procedure to the complete HST T.

▶ **Lemma 13.** *The above splitting procedure does not change the hierarchical block partition and it thus also preserves* Arrow*'s queueing order $\pi_{\mathcal{A}}$ and its total cost $cost_{\mathcal{A}}(\pi_{\mathcal{A}})$.*

The next lemma shows that if a tree T'' is split into two trees T_1'' and T_2'' such that all requests in T_1'' are ordered before all requests in T_2'', there is a significant time of occurrence gap between the requests ending up in subtrees T_1'' and T_2''.

▶ **Lemma 14.** *Assume that we are performing a single splitting. Further, assume that we are working on a tree T' on level ℓ and that we are splitting a subtree T'' of T' into T_1'' and T_2'' such that T_1'' obtains the blocks that are scheduled first by* Arrow*. If R_1 and R_2 are the request sets of T_1'' and T_2'', respectively, we have $t_{\min}(R_2) - t_{\max}(R_1) \geq \delta(\ell) - \delta(\ell-1)$.*

It remains to show that the splitting also does not affect the optimal offline cost in a significant way. The following lemma shows that the Manhattan cost $c_{\mathcal{M}}(r, r')$ for any two requests r and r' can increase by at most a factor 3. Hence, also the total Manhattan cost of an optimal ordering cannot increase by more than a factor 3.

▶ **Lemma 15.** *For any two requests r and r', the splitting procedure does not increase the Manhattan cost $c_{\mathcal{M}}(r, r')$ by more than a factor 3.*

For the remainder of the analysis, we assume that the HST T is an HST that is obtained after applying the splitting procedure recursively. We therefore assume that for every level ℓ and every subtree T' at level ℓ, there is no level-$(\ell-1)$ subtree T'' of T' that contains two neighbor blocks that satisfy Condition (6).

3.3 Lower Bounding The Optimal Manhattan Cost

In this section, we construct a tree S^* that spans all requests in R. The tree S^* has a nice hierarchical structure: For each subtree T' of T, the set of edges of S^* induced by the request set of the subtree T' forms a spanning tree of the request set of T'. Apart from this useful structural property, we will show that the total Manhattan cost of the spanning tree S^* is within a constant factor of minimum spanning tree (MST) of the request set R w.r.t. the Manhattan cost. We have seen that on condensed request sets, the optimal TSP path of the request set w.r.t. the Manhattan cost is within a constant factor of the optimal offline queueing cost. Note that because any TSP path is also a spanning tree, this implies that the total Manhattan cost of the MST and thus also the total Manhattan cost of the tree S^* are lower bounding the optimal offline queueing cost within a constant multiplicative factor.

For convenience, we add one more level to the HST T. Instead of placing the requests at the leaves on level 0, we assume that each level 0 node v has a child node on level -1 for each of the requests issued at node v. Hence, the new leaf nodes are on level -1 and each leaf node receives exactly one request.[4] The distance between a level -1 node and its parent on level 0 is set to be 0.

Spanning Tree Construction. The spanning tree S^* is constructed greedily in a bottom-up fashion. For each subtree T' of T, we recursively define a tree $S^*(T')$ as follows. For the leaf nodes on level -1, the tree consists of the single request placed at the node. For a tree T' rooted at a node v on level $\ell \geq 0$, the tree $S^*(T')$ consists of the recursively constructed trees $S^*(T_1''), S^*(T_2''), \ldots$ of the subtrees $T_1'', T_2'', \ldots$ of T'' and of edges connecting the trees $S^*(T_1''), S^*(T_2''), \ldots$ to a spanning tree of the set of requests issued at leaves of tree T'. The edges for connecting the trees $S^*(T_1''), S^*(T_2''), \ldots$ are chosen so that they have minimum total Manhattan cost. That is, to connect the trees $S^*(T_1''), S^*(T_2''), \ldots,$ we compute an MST of the graph we get if each of the trees $S^*(T_i'')$ is contracted to a single node. We can therefore for example choose the edges to connect the trees $S^*(T_1''), S^*(T_2''), \ldots$ in e greedy way: Always add the lightest (w.r.t. Manhattan cost) edge that does not close a cycle with the already existing edges, including the edges of the trees $S^*(T_1''), S^*(T_2''), \ldots.$

MST Approximation. In the following, it is shown that the total Manhattan cost of the tree $S^* = S^*(T)$ is within a constant factor of the cost of an MST w.r.t. the Manhattan cost. Where convenient, we identify a tree τ with its set of edges, i.e., we also use S^* to denote the set of edges of the tree S^*. Further, the cost of an edge $e = \{r, r'\}$ is the Manhattan cost $c_{\mathcal{M}}(r, r')$. We also slightly abuse notation and use $c_{\mathcal{M}}(e)$ to denote this cost. The proof applies a general MST approximation result that appears as Theorem A.1 in the full version [11]. Together with the following lemma, Theorem A.1 of [11] directly implies that the total Manhattan cost of S^* is within a factor 4 of the MST Manhattan cost. For a subtree T' of T, we use $R(T')$ to denote the subset of the requests in R that are issued at nodes of T'.

[4] Note that subtrees of T that do not have any queueing requests can be ignored and therefore, we can w.l.o.g. assume that every leaf node issues some queueing request.

▶ **Lemma 16.** *Consider the constructed spanning tree S^* and consider an arbitrary edge e of S^*. Let S_1^* and S_2^* the two subtrees that result when removing edge e from S^*. Further, assume e^* be an edge that connects the two subtrees S_1^* and S_2^* and that has minimum Manhattan cost among all such edges. We then have $c_{\mathcal{M}}(e) \leq 4 \cdot c_{\mathcal{M}}(e^*)$.*

▶ **Corollary 17.** *The total Manhattan cost of the spanning tree S^* is at most 4 times the total Manhattan cost of an MST spanning all the requests.*

4 Analysis of the Online Queueing Cost

In this section, we give a general framework to compare the queueing cost of an online queueing algorithm on HST T with the bound of the offline queueing cost as established in Section 3. At the end of the section, we apply the method to analyze synchronous Arrow executions on T. As in Section 3.3, for convenience, we add one more level to the HST T so that each level 0 node v has a child node on level -1 for each of the requests issued at node v. The new leaf nodes are on level -1 and each leaf node receives exactly one request.

We first state two basic locality properties of Arrow. We will then show that those properties are sufficient to prove a constant competitive ratio compared to the optimal offline queueing cost on T. We define the notion of a *distance-respecting queueing order* and the notion of *distance-respecting latency cost* of a queueing algorithm.

▶ **Definition 18** (Distance-Respecting Order)**.** Let R be a set of requests $r_i = (v_i, t_i)$ issued at the nodes of a tree T and let π be permutation on $[0, |R| - 1]$. The ordering $r_{\pi(0)}, r_{\pi(1)}, \ldots, r_{\pi(|R|-1)}$ induced by π is called *distance-respecting* if whenever $\pi(i) < \pi(j)$, we have $t_i - t_j \leq d_T(v_i, v_j)$.

▶ **Definition 19** (Distance-Respecting Latency Cost)**.** An online distributed queueing algorithm ALG is said to have *distance-respecting latency cost* if for any request set R and any possible queueing order π_{ALG} of ALG, for all $1 \leq i < j < |R|$, it holds that

$$t_{\pi_{\mathsf{ALG}}(i)} + L_{\mathsf{ALG}}(r_{\pi_{\mathsf{ALG}}(i)}, \pi_{\mathsf{ALG}}(i-1)) \leq t_{\pi_{\mathsf{ALG}}(j)} + d_T(v_{\pi_{\mathsf{ALG}}(j)}, v_{\pi_{\mathsf{ALG}}(i-1)}).$$

4.1 Constructing a Spanning Tree

As the first part of the online queueing cost analysis, we construct a new tree $\mathbb{S}$ that spans all requests in R. It will be shown that the total Manhattan cost of $\mathbb{S}$ asymptotically equals the total Manhattan cost of the tree S^* constructed in the previous section.

We construct a new tree $\mathbb{S}$ on R based on an ordering π of the set of requests. We assume that the ordering of the requests given by π is $r_{\pi(0)}, r_{\pi(1)}, \ldots, r_{\pi(|R|-1)}$. For each index i with $i \in [0, |R| - 2]$, we define the *local successor* as

$$next(i) := \min \left\{ j \in [i+1, |R|-1] \,:\, d_T(v_{\pi(i)}, v_{\pi(j)}) = \min_{k \in [i+1, |R|-1]} d_T(v_{\pi(i)}, v_{\pi(k)}) \right\}. \quad (7)$$

Hence, among the requests ordered after $r_{\pi(i)}$ by order π, $next(i)$ is the position of a request in the order π with minimum tree distance to $v_{\pi(i)}$ and among those, of the first one ordered by π. Note that this means that for all requests $r_{\pi(k)}$ for which $i < k < next(i)$, we have $d_T(v_{\pi(i)}, v_{\pi(k)}) > d_T(v_{\pi(i)}, v_{\pi(next(i))})$ and for all requests $r_{\pi(k)}$ for which $k \geq next(i)$, we have $d_T(v_{\pi(i)}, v_{\pi(k)}) \geq d_T(v_{\pi(i)}, v_{\pi(next(i))})$.

The spanning tree $\mathbb{S}$ is constructed as follows. For every request $r_{\pi(i)}$ for all $i \in [0, |R| - 2]$, we add the edge $\{r_{\pi(i)}, r_{\pi(next(i))}\}$ to the tree $\mathbb{S}$. Note that $\mathbb{S}$ is indeed a spanning tree: If

directing each edge from $r_{\pi(i)}$ to $r_{\pi(next(i))}$, each node has out-degree 1 and we cannot have cycles because $next(i) > i$. The following observation shows that in addition, $\mathbb{S}$ has the same useful hierarchical structure as the tree S^* constructed in Section 3.3.

▶ **Observation 20.** *As the tree S^*, also the tree $\mathbb{S}$ has the property that for any subtree T' of T, the subgraph of $\mathbb{S}$ induced by only the requests at nodes in T' is a connected subtree of $\mathbb{S}$. This follows directly from the definition of the local successor $r_{\pi(next(i))}$. Except for the last ordered request inside T', the local successor of any other request of T' is inside T' (because the local successor is a request with minimum tree distance).* ◀

In light of Observation 20, for any subtree T' of T, we use $\mathbb{S}(T')$ to denote the subtree of $\mathbb{S}$ induced by the requests issued at nodes in T'.

4.2 Bounding the Manhattan Cost of the Spanning Tree

The following lemma shows that if the spanning tree $\mathbb{S}$ is constructed by using a distance-respecting ordering π, the total Manhattan cost of the spanning tree $\mathbb{S}$ is asymptotically equal the total Manhattan cost of S^*.

▶ **Lemma 21.** *Let $C_{\mathcal{M}}(\mathbb{S})$ and $C_{\mathcal{M}}(S^*)$ be the total Manhattan costs of $\mathbb{S}$ and of S^*. If the tree $\mathbb{S}$ is constructed using a distance-respecting ordering π, we have $C_{\mathcal{M}}(\mathbb{S}) \leq 3 \cdot C_{\mathcal{M}}(S^*)$.*

4.3 Bounding the Total Latency Cost

It remains to prove the main claim and show that the total online queueing cost on the HST T is within a constant factor of the optimal offline cost on T. The following theorem states that this is generally true for algorithms with distance-respecting latency cost (Def. 19) and which produce distance-respecting queueing orders (Def. 18), as long as the request set R is condensed (Def. 4).

▶ **Theorem 22.** *Assume that we are given an HST T and a condensed set of requests issued at the leaves of R. Further, assume that we are given a distributed queueing algorithm* ALG *that has distance-respecting latency cost and that always produces a distance-respecting queueing order π. Then, the total latency cost of* ALG *is within a constant factor of the optimal offline cost on T.*

▶ **Corollary 23.** *The total latency cost of a synchronous execution of* Arrow *on an HST T is within a constant factor of the optimal offline queueing cost on T.*

▶ Remark. The above corollary proves Theorem 1 (cf. Section 1) for synchronous executions on the HST T. The full statement of Theorem 1 for general asynchronous executions is proven in the full version of the paper [11]. There, it is shown that also for asynchronous executions, Arrow has distance-respecting latency cost and produces distance-respecting queueing orders. In addition, we also show that we can still restrict attention to condensed request sets. The claim of Theorem 1 for the asynchronous case then follows from Theorem 22 in the same way as in the above corollary.

References

1 I. Abraham, D. Dolev, and D. Malkhi. LLS: a locality aware location service for mobile ad hoc networks. In *D-POMC co-located 10th C. Mobicom*, pages 75–84, 2004.

2 H. Attiya, V. Gramoli, and A. Milani. A provably starvation-free distributed directory protocol. In *Proc. of the 12th Symp. on Self-Stabilizing Syst. (SSS)*, pages 405–419, 2010.

3 B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st Symp. Foundations of Computer Science (FOCS)*, pages 503–513, 1990.

4 Baruch Awerbuch and David Peleg. Online tracking of mobile users. *Journal of the ACM (JACM)*, 42(5):1021–1058, 1995.

5 Y. Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *Proc. 37th Symp. on Foundations of Computer Science (FOCS)*, pages 184–193, 1996.

6 D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, 1990.

7 M. Demirbas et al. A hierarchy-based fault-local stabilizing algorithm for tracking in sensor networks. In *P. 8th I. C. on Princ. of Dist. Syst. (OPODIS)*, pages 299–315, 2004.

8 Michael J Demmer and Maurice P Herlihy. The arrow distributed directory protocol. In *Proc. of the 12th Symp. on Dist. Comp. (DISC)*, pages 119–133, 1998.

9 J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proc. 35th S. on Th. of Comp. (STOC)*, pages 448–455, 2003.

10 Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Dist. Comp.*, pages 197–211, 2014.

11 Abdolhamid Ghodselahi and Fabian Kuhn. Dynamic analysis of the arrow distributed directory protocol in general networks. *arXiv preprint arXiv:1705.07327*, 2017.

12 David Ginat, Daniel D Sleator, and Robert E Tarjan. A tight amortized bound for path reversal. *Information Processing Letters*, 31(1):3–5, 1989.

13 A. Gupta. Steiner poi,nts in tree metrics don't (really) help. In *Proc. 12th Symp. on Discrete Algorithms (SODA)*, pages 220–227, 2001.

14 M. Herlihy, F. Kuhn, S. Tirthapura, and R. Wattenhofer. Dynamic analysis of the arrow distributed protocol. *Theory of Comp. Syst. (TCS)*, 39(6):875–901, 2006.

15 M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proc. of the 20th Symp. on Princ. of Dist. Comp. (PODC)*, pages 127–133, 2001.

16 Maurice Herlihy. The Aleph toolkit: Support for scalable distributed shared objects. In *Proc. 3rd Workshop on Comm. Arch. and Appl. for Network-Based Parallel Comp. (CANPC)*, pages 137–149, 1999.

17 Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

18 Maurice Herlihy, Srikanta Tirthapura, and Rogert Wattenhofer. Ordered multicast and distributed swap. *ACM SIGOPS Operating Syst. Rev. (OSR)*, 35(1):85–96, 2001.

19 Maurice Herlihy and Michael P Warres. A tale of two directories: implementing distributed shared objects in java. In *Proc. of the ACM Conf. on Java Grande*, pages 99–108, 1999.

20 Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Syst. (TOCS)*, 7(4):321–359, 1989.

21 Mohamed Naimi and Michel Trehel. An improvement of the $\log n$ distributed algorithm for mutual exclusion. In *Proc. 7th Conf. on Distr. Comp. Sys. (ICDCS)*, pages 371–377, 1987.

22 D. Peleg and E. Reshef. A variant of the arrow distributed directory with low average complexity. In *Proc. of the 26th Int. Coll. on A. L. and P. (ICALP)*, pages 615–624, 1999.

23 Y. Rabinovich and R. Raz. Lower bounds on the distortion of embedding finite metric spaces in graphs. *Discrete and Computational Geometry*, (19), 1998.

24 Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. on Computer Syst. (TOCS)*, 7(1):61–77, 1989.

25 R. Rosenkrantz, R. Stearns, and P. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM J. on Computing*, 6(3):563–581, 1977.

26 Gokarna Sharma and Costas Busch. Distributed transactional memory for general networks. *Distributed Computing*, 27(5):329–362, 2014.

27 Gokarna Sharma and Costas Busch. An analysis framework for distributed hierarchical directories. *Algorithmica*, 71(2):377–408, 2015.

28 Srikanta Tirthapura and Maurice Herlihy. Self-stabilizing distributed queuing. *IEEE Trans. on Parallel and Dist. Syst. (PDS)*, 17(7):646–655, 2006.

29 Jan L. A. van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2(2):113–115, 1987.

30 B. Zhang and B. Ravindran. Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In *Proc. of the 24th IPDPS*, pages 1–11, 2010.