Competitive Concurrent Distributed Queuing

Maurice Herlihy, Brown University, Providence RI 02912, mph@cs.brown.edu
Srikanta Tirthapura, Brown University, Providence RI 02912, snt@cs.brown.edu
Roger Wattenhofer, Microsoft Research, Redmond WA 98052, rogerwa@microsoft.com

May 7, 2001

Abstract

Distributed queuing is a fundamental problem in distributed computing, arising in a variety of applications. The challenge in designing a distributed queuing algorithm is to minimize message traffic and delay.

This paper gives a novel competitive analysis of the Arrow distributed queuing protocol under concurrent access. We analyze the combined latency of r simultaneous requests, and derive a competitive ratio of $s \cdot \log r$, where s is the stretch of a preselected spanning tree in the network.

Our analysis employs a novel *greedy* characterization of the way the Arrow protocol orders concurrent requests, and yields a new lower bound on the quality of the nearest-neighbor heuristic for the Traveling Salesperson Problem.

1 Introduction

In the *distributed queuing* problem, processes in a message-passing network asynchronously and concurrently place themselves in a distributed logical queue. Specifically, each participating process informs its predecessor of its identity, and (when appropriate) learns the identity of its successor. The challenge in designing a distributed queuing protocol is to minimize message traffic and delay.

Distributed queuing is a fundamental problem in distributed computing, arising in a variety of distributed applications. For example, we have used the Arrow distributed queuing protocol [4] as the basis for managing mobile objects in the Aleph Toolkit [5], a distributed shared object system that provides transparent caching and synchronization of mobile objects. Experimental results show that the Arrow protocol substantially outperforms conventional home-based schemes under high contention [7].

Distributed queuing can also be used for distributed mutual exclusion (by passing a token along the queue), distributed counting (by passing a counter), or distributed implementations of synchronization primitives such as swap. We have shown how to use distributed queuing for scalable ordered multicast in [6].

The Arrow protocol [4] is a simple distributed queuing protocol based on path reversal in a network spanning tree (we give an informal description of the protocol in section 2). This paper gives a novel competitive analysis of the performance of the Arrow protocol under concurrent access. A queuing algorithm has many options for handling concurrent requests. For example, when presented with simultaneous requests from nodes a, b, and c, where a and b are near one another but c is far, it makes sense to avoid ordering c between a and b. More generally, r concurrent requests can be ordered in any of $r!$ ways, and depending on the origins of the requests, some orderings may be much more efficient than others. A common way to evaluate the effectiveness of an on-line distributed algorithm is to compare its performance to an optimal off-line algorithm (or "adversary"), one that pays nothing for synchronization and can make routing decisions based on "omniscient" global information.

Prior work: In the original paper describing the Arrow protocol, Demmer and Herlihy [4] give a competitive analysis under sequential access. Sequential access assumes that the Arrow protocol and the adversary would queue requests in the same order. Arrow would send a message from a node to its predecessor on the shortest (and only) path on the spanning tree while the optimal protocol could do so on the shortest path on the graph. The *stretch* s of a tree T is the worst-case ratio between the shortest paths linking two vertices in T and the same vertices in G. Thus the Arrow protocol has a worst case competitive ratio of s in the sequential case. Peleg and Reshef [8] show that if one knows something about the probability distributions of requests at each node, it is possible to choose a spanning tree on which the expected overhead of the Arrow protocol is small.

The above analyses do not apply to concurrent access because the adversary is not allowed to gain by ordering concurrent (or nearly concurrent) requests in a smarter way. In this paper, we study the following *one-shot* instance of the problem for concurrent access. At time zero, r nodes issue requests to join the queue (and no further requests occur). A request's *la-*

tency is the time needed for the originating node to inform its predecessor of its identity. A natural measure of work is the *sum* of all requests' latencies. Our analysis is based on a synchronous network in which nodes do not crash, messages are not lost, and each communication link has a fixed latency.

For r concurrent requests, we derive a competitive ratio of $s \cdot \log r$. This ratio does not depend on the size of the network, and depends only logarithmically on the degree of concurrency. Informally, the adversary could win over a distributed protocol like the Arrow protocol in two ways: (1)It could communicate over the graph, while the Arrow protocol communicates over a tree, since it needs to synchronize the requests. This leads to the factor of stretch in the competitive ratio. (2)It could select the queuing order of the requests in an optimal way, while the distributed protocol, with local information could be sub-optimal in its ordering. For the arrow protocol, this results in a factor of $\log r$ in the competitive ratio. From a practical perspective, we would like to point out that there are no hidden constants in the competitive ratio.

Our analysis employs a novel *greedy* characterization of how Arrow orders concurrent requests, and yields an intriguing connection with the nearest neighbor heuristic for the Traveling Salesperson Problem (TSP). A further contribution of this paper is a new lower bound on the quality of a nearest-neighbor TSP algorithm on a tree. Rosenkrantz, Stearns, and Lewis [9] have shown that the nearest neighbor algorithm is a $\log r$ approximation algorithm for TSP on a graph with r nodes which satisfies the triangle inequality. Since a tree metric obeys the triangle inequality, this result implies the same $\log r$ upper bound for the nearest neighbor algorithm over a tree metric. We show that there exist tree metrics on which the greedy algorithm could be off by as much as a factor of $\Omega(\log r / \log \log r)$ from optimal. It follows that a graph having the stronger tree metric property does not substantially improve the behavior of the nearest-neighbor TSP heuristic.

The rest of the paper is organized as follows. Section 2 defines the problem formally and explains the model of computation. Section 3 gives the competitive analysis of the Arrow protocol for the one-shot problem. Section 4 contains a discussion of the competitive ratio and we end with the open problems and conclusions.

2 Model and Problem Description

We first give an informal presentation of the Arrow protocol (more detailed descriptions appear elsewhere [4, 6]). The protocol runs on a fixed spanning tree T of the network graph G. Each node stores an "arrow" which can point either to itself, or to any of its neighbors in T. The meaning of the arrow is the following: if a node's arrow points to itself, then that node is tentatively the last node in the queue. Otherwise, if the node's arrow points to a neighbor, then the end of the queue currently resides in the component of the directory tree containing that neighbor. Informally, except for the node at the end of the queue, a node knows only in which "direction" the end of the queue lies.

The protocol is based on path reversal. Initially, one node is selected to be the head of the queue, and the tree is initialized so that following the arrows from any node leads to that head. To place itself on the queue, a node v sends a $find(v)$ message to the node indicated by its arrow, and "flips" its arrow to point to itself. When a node x whose arrow points to u receives a $find(v)$ message from tree neighbor w, it immediately "flips" its arrow back to w. If x is not in the queue, it forwards the message to u, the prior target of its arrow. If x is in the queue, then it has just learned that v is its successor. (In many applications of distributed queuing, x would then send a message to v, but we do not consider that message as a part of the queuing protocol itself.) In [4], the authors prove the correctness of the protocol in an asynchronous model. They also show that $find(v)$ travels on a simple path on T from v to its predecessor.

We model the network as a graph $G = (V, E)$ where V is the set of nodes (processors), and E is the set of edges, representing reliable FIFO communication links between processors. Each edge e has a weight $w(e)$ equal to the latency of the communication link. The Arrow protocol runs on a spanning tree T of this graph, which we can choose. Denote the length of the shortest path between nodes u and v on G by $d_G(u, v)$ and the length of the shortest path between them on T by $d_T(u, v)$. The *stretch* of the tree T with respect to G is defined as $s = \max_{u,v \in V} d_T(u, v)/d_G(u, v)$. This ratio measures how far from optimal a path through the spanning tree T can be.

We assume a synchronous model for our analysis (although the protocol does not require synchrony for correctness). A $find()$ message arriving at a node is processed immediately, and simultaneously arriving messages are processed in an arbitrary order. For a simple example with concurrent $find()$ operations, see Figure 1. In practice, the time needed to service a message is small compared to communication latencies, and because the

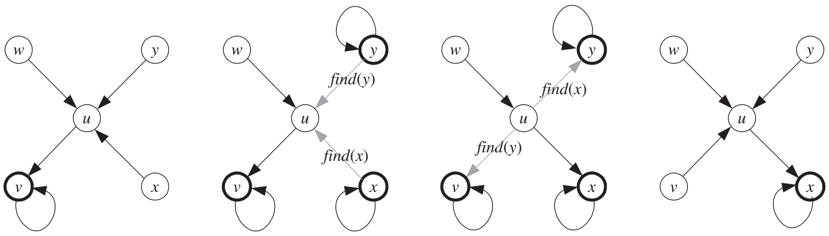


Figure 1: Concurrent $find$ messages.

Initially node v is selected to be the head of the queue. Nodes x and y both place themselves on the queue. Message $find(y)$ arrives at node u before $find(x)$. Finally, $find(x)$ and $find(y)$ find their respective predecessors y and v in the queue, and x is the new head of the queue.

degree of the spanning tree is typically small, a node cannot receive very many simultaneous messages.

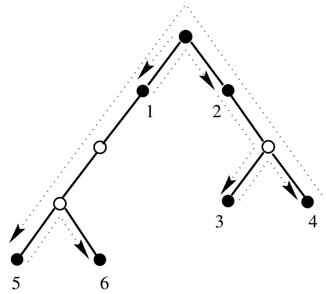
In the *one-shot* problem, at time zero, a subset $R \subseteq V$ of nodes request to join the queue, with $r = |R|$. Let $l_A(v)$ be the latency of node v to find its predecessor using the Arrow protocol. The cost of the Arrow protocol is $L^A = \sum_{v \in R} l_A(v)$. We compare L^A to a lower bound L^*, which is the cost of an optimal protocol (with global knowledge and incurring no synchronization costs). The competitive ratio of Arrow is defined to be $\max_{R \subseteq V}\{L^A/L^*\}$.

3 Competitive Analysis

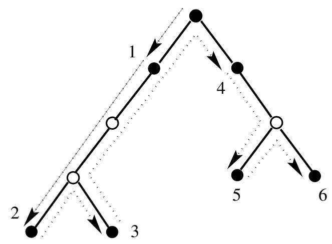
In this section, we present an upper bound for L^A and a lower bound for L^*. We first analyze L^A.

Suppose the $find()$ operations are executed in the order $v_1, v_2, \ldots, v_r$, that is, *root* (the initial head of the queue) is the predecessor of v_1, which precedes v_2 and so on. Note that the order of execution is not necessarily the real time order of completion. All the queuing is done in parallel, so v_2 might learn about its successor v_3 before v_1 learns about v_2.

As shown in [4] a $find()$ travels along a simple path on the tree until it finds the predecessor, and never waits. Hence $l_A(v_1) = d_T(v_1, root)$, and $l_A(v_i) = d_T(v_i, v_{i-1})$, for $i > 1$. We have:



(a) A greedy walk on a tree.



(b) An Euler tour traversal of the same tree.

Figure 2: A greedy walk on a tree vs Euler tour
Vertices belonging to R are marked solid; every edge has weight 1. The numbers at the vertices indicate the order in which they are visited. The traversals start at the root, which is the topmost vertex.

$$L^A = d_T(v_1, root) + \sum_{i=2}^{r} d_T(v_i, v_{i-1}). \tag{1}$$

We now give a simple characterization of the order $v_1 \ldots v_r$. A greedy walk on tree T over vertex set $R = \{u_1, \ldots, u_r\}$ visits all vertices in R as follows: It starts at the root of the tree. It then visits the closest unvisited vertex in R, and keeps doing so until all vertices in R have been visited. In other words, the vertices of R are visited in the order $v_1, \ldots, v_r$ with

$$d_T(root, v_1) = \min_{v \in R} d_T(root, v) \tag{2}$$

$$d_T(v_i, v_{i+1}) = \min_{v \in R} d_T(v_i, v) \text{ with } v \notin \{v_1, \ldots, v_i\} \tag{3}$$

An example of a greedy walk is shown in Figure 2. An ordering of vertices is *greedy* if there is a greedy walk that produces the same ordering. Denote the length of a greedy walk on tree T over vertex set R by $greedy(T, R)$.

Theorem 1 *The ordering of the Arrow protocol is greedy, in other words, the ordering of the requests satisfies equations 2 and 3.*

Proof: We first prove Equation 2. Let C be the set of all the closest requests to the root, and let d be the distance between them and the root at time 0. At time 0, requests in C start traveling towards the root, since the tree is initialized with arrows pointing towards the root. If two (or more) of these $find()$ requests meet at a node, then one continues towards the root and the others are deflected. Therefore at least one of the requests in C arrives at the root at time d. Since no request outside C can reach the root in time d or less, we have $v_1 \in C$, as in equation 2.

We now prove Equation 3. Denote the root by v_0. Consider another starting configuration of the distributed system, where the tree is initialized with v_1 as the root and there is no request at v_1. Call this configuration F_1 and the original one (with v_0 as the root) F_0.

Lemma 2 *No request but v_1 will be able to distinguish between the configurations F_0 and F_1 during execution.*

Proof: Refer to Figure 3. The only difference between F_0 and F_1 is that all arrows on the path between v_0 and v_1 are in the opposite direction. Assume, for the sake of contradiction, that there is a $find(v_i)$ with $i > 1$ that is able to see an arrow pointing towards v_0 before $find(v_1)$ changes it. Then $find(v_i)$ must reach a node u (u between v_0 and v_1) before $find(v_1)$. If so, $find(v_1)$ would be deflected towards v_i, which is a contradiction to the assumption that v_1 is the first request in the total order. ■

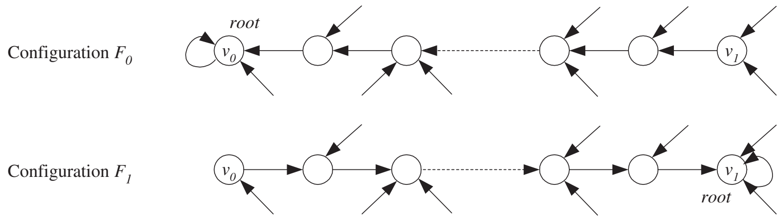


Figure 3: The two configurations are identical to every request but v_1.

Lemma 2 implies that for the purpose of finding the ordering of $\{v_i | i > 1\}$ we can pretend as if we started in configuration F_1 and $find(v_2)$ will find v_1 in the resulting execution. Applying Equation 2 to F_1, we find that v_2 is one of the requests closest to v_1. Inductively, we define F_i to be the configuration derived from F_0 by removing the requests at nodes $v_1, \ldots, v_i$, and making v_i the root of the tree. For the rest of the requests $\{v_j | j > i\}$, F_i is identical to F_0. Equation 3 follows. This concludes the proof of Theorem 1. ■

The above Theorem lets us relate L^A to the cost of traveling salesperson tours. Let $TSP(G)$ denote the cost of the optimal traveling salesperson tour on graph G.

Theorem 3 *Let G_R^T be the complete graph with vertices R (plus the root) and distance between vertices u and v equal to $d_T(u,v)$. Then $greedy(T,R) \leq \log r \cdot TSP(G_R^T)/2$.*

Proof: The Theorem follows directly from Theorem 1 of Rosenkrantz, Stearns, and Lewis [9]. They show that the nearest neighbor heuristic is at most a factor logarithmic in the number of nodes worse than an optimal traveling salesperson tour on a graph satisfying certain conditions. A greedy walk costs exactly as much as the nearest neighbor heuristic without returning to the root.

We note that the graph G_R^T satisfies the three preconditions of the Theorem, that is, $d(u,v) = d(v,u)$, $d(u,v) \geq 0$, and the triangle inequality $d(u,v) + d(v,w) \geq d(u,w)$. ■

Let T_R denote the smallest subtree of T containing all the vertices in R and the root. Note that the optimal TSP on G_R^T corresponds to an Euler tour traversal of T_R, as shown in Figure 2, and then returning back to the root.

Theorems 1 and 3 immediately lead to the following corollary.

Corollary 4 $L^A \leq \log r \cdot TSP(G_R^T)/2$.

Theorem 5 *Let G_R be a complete graph on R (plus the root) with the weight of the edge between two nodes u and v equal to $d_G(u,v)$. Then $L^* \geq TSP(G_R)/2$.*

Proof: Let the set of request nodes be $R = \{u_1, \ldots, u_r\}$, and assume that the optimal algorithm orders the requests as $u_1, \ldots, u_r$. Since u_i's queuing is complete only when u_{i-1} learns of the identity of u_i, the latency of u_i's request is at least $d_G(u_{i-1}, u_i)$ (and the latency of u_1's request is at least $d_G(root, u_1)$). Thus, the sum of the latencies of the optimal algorithm is at least

$$L^* \geq d_G(root, u_1) + \sum_{i=2}^{r} d_G(u_i, u_{i-1}).$$

When finally returning to the root, we have a valid (but not necessarily optimal) TSP tour with cost $C \leq L^* + d_G(u_r, root)$. The graph G_R satisfies the triangle inequality, since its edge weights are lengths of shortest paths between the vertices on G. Thus, $d_G(u_r, root) \leq L^*$, and therefore $TSP(G_R) \leq C \leq 2L^*$. ∎

Corollary 6 $L^A/L^* \leq s \cdot \log r$.

Proof: The edge between vertices u and v in G_R^T has weight $d_T(u, v)$ while the corresponding edge in G_R has weight $d_G(u, v)$. The edge on G_R^T can be longer than the corresponding edge in G_R by a factor of at most s. The same ratio carries over to the length of the optimal TSP tours and we have $TSP(G_R^T) \leq s \cdot TSP(G_R)$.

With Corollary 4 and Theorem 5 we get

$$\begin{aligned} L^A &\leq \log r \cdot TSP(G_R^T)/2 \\ &\leq \log r \cdot s \cdot TSP(G_R)/2 \\ &\leq \log r \cdot s \cdot L^*. \end{aligned}$$

∎

In the remainder of this section we show that our analysis is (almost) tight. We will construct a tree, along with a set of requesting nodes where the greedy (nearest neighbor) walk is off by $\Omega(\log r/\log\log r)$ from optimal. It follows that having the tree metric does not help (much) over the more general triangle inequality metric. To the best of our knowledge, this is a new result in the area of TSP heuristics as well.

Theorem 7 *There exists a tree T and a set of requesting nodes R such that $L^A = \Omega(\log r/\log\log r)L^*$, where $r = |R|$.*

Proof: The tree T consists of a long "trunk" with many "branches" of varying lengths on it as shown in the figure 4. Each branch is a single edge; one end of the edge is on the trunk and the other end is a leaf with a request on it. A branch of length 0 is a request on the trunk. The root is at one end of the trunk and the other end is denoted by $\top$. Our convention is that we move *right* to get from the *root* to $\top$. The distance between two branches is the distance between the endpoints of the branches which are lying on the trunk. Similarly, the distance between a vertex v on the trunk and a branch e is the distance between v and the endpoint of e that lies on the trunk.

The idea is as follows: by careful placement of branches on the trunk, we will make the greedy walk traverse the length of the trunk many times, as shown in figure 5. The trunk contributes a significant fraction of the weight

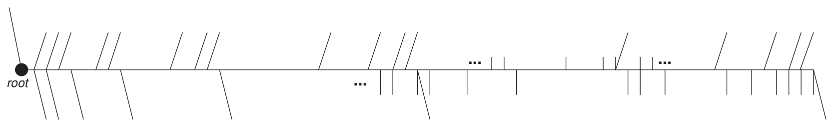


Figure 4: Tree T

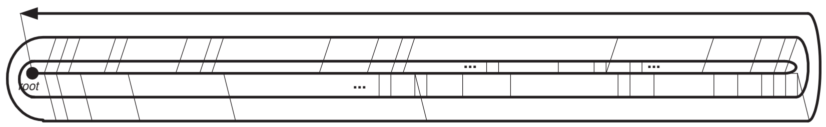


Figure 5: Greedy Walk on Tree T

of the tree, and we get the length of the greedy walk to be super-linear in the size of the tree and hence the length of the Euler tour. The details follow.

Let the length of the trunk be w. Let $k = \log w / \log\log w$ rounded down to the nearest odd number. We have $k+1$ sets of branches, $B_0 \ldots B_k$. Each branch in B_i is of length i. Thus requests in B_0 are on the trunk, while those in B_1 are at distance 1 from the trunk, and so on. When the context is clear, we use B_i to refer to the set of requests that lie on the branches in B_i.

There is only one branch in B_k and this is at the root. Once we have placed all the branches in B_j, we place those in B_{j-1} as follows.

Suppose j was odd. Let $e_1, e_2 \in B_j$ be two consecutive branches in B_j starting at vertices u_1 and u_2 from the trunk respectively (i.e there are no branches in B_j with endpoints between u_1 and u_2). Suppose u_1 is closer to the root than u_2. Let l be the least integer such that $2^{l+1} \geq d_T(u_1, u_2)$. We place branches in B_{j-1} at vertices between u_1 and u_2 at geometrically increasing distances $1, 3 \ldots 2^l - 1$ from u_1. This is shown in figure 6. If the farthest branch from the root in B_j starts at u, then we place branches in B_{j-1} between u and $\top$ at distances $1, 3 \ldots 2^l - 1$ from u until $2^{l+1} > d(u, \top)$. We also place a branch in B_{j-1} at $\top$.

Suppose that j was even. The construction is similar to the above, but the role of $\top$ and the root are interchanged. In other words, if $e_1, e_2 \in B_j$ are two consecutive branches in B_j starting at vertices u_1 and u_2 from the trunk respectively and suppose u_1 is closer to the root than u_2. Let l be the least integer such that $2^{l+1} \geq d_T(u_1, u_2)$. We place branches in B_{j-1} at vertices between u_1 and u_2 at distances $1, 3 \ldots 2^l - 1$ from u_2 (not u_1).

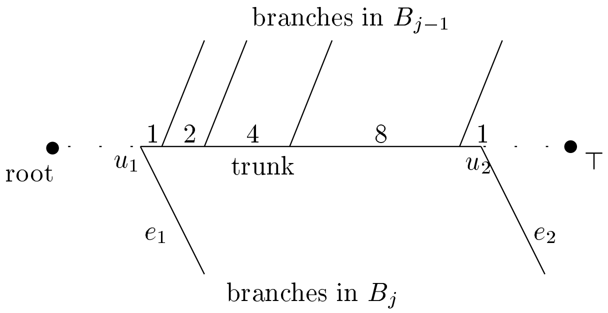


Figure 6: Placement of branches in B_{j-1} between two branches in B_j

Similarly, we place branches in B_{j-1} between the branch in B_j that is closest to the root and the root, and we place a branch at the root.

Lemma 8 *For any vertex on the trunk, one of the closest requests in the set of branches $\{\cup_{i\geq c} B_i\}$ is in B_c.*

Proof: Let x be a vertex on the trunk. We show that the distance to the closest request in B_i is lesser than or equal to the distance to the closest request in B_{i+1}. Suppose r_p was the closest request in B_{i+1}, whose branch starts from the trunk at vertex p.

Consider the following case: p is to the left of x (or $p = x$) and i is even (or zero). There is a branch in B_i to the right of p at distance 1 from p (if p is $\top$, then there is a branch in B_i at $\top$). The request on this branch is certainly closer (or the same distance as) to x than r_p.

The other cases, p on the left of x and i odd, and the analogous cases for p to the right of x can be checked similarly. ∎

Theorem 9 *The following is a greedy walk on T. Start from the root. Visit all requests in order of the branch size (i.e. all requests in B_0 first, followed by those in B_1 and so on until B_k). If i is even (or zero), visit the requests in B_i in order of increasing distance from the root. If i is odd, then visit them in order of increasing distance from $\top$.*

Proof: We show that visiting all the requests in B_0 in order of increasing distance from the root is a prefix of a greedy walk. This portion of the walk ends at $\top$. Then the proof follows, since after that we can treat $\top$ as the root, and it is a similar situation.

Clearly, the first request visited is the closest request in B_0 because of Lemma 8. Suppose we are at vertex x on the trunk. All the requests in B_0

to the left of x have been visited. None to the right have been. Let c_0 be the closest request in B_0 that is to the right of x. We will show that c_0 is one of the closest unvisited requests. Going to c_0 next would be greedy, and this way we visit all the requests in B_0 in order of increasing distance from the root and reach $\top$.

To prove that c_0 is one of the closest unvisited requests, we first show that c_0 is not further away from x than the closest request in B_1 (say c_1). By Lemma 8, one of the closest requests to x in the set $\{\cup_{i\geq 1} B_i\}$ is in B_1, and the proof follows.

We now show that c_0 is not further away from x than c_1. Note that the request c_1 could be to the right or left of x, but c_0 is to the right of x. Suppose c_1 was to the right of x. Recall that in our construction, to the right of every branch in B_1, there is a branch in B_0 at a distance of 1 (or if c_1 is $\top$, then there's a branch in B_0 at $\top$). The request on this branch in B_0 is at least as close to x as c_1. Now suppose that c_1 was to the left of x. There are branches in B_0 at distances $1, 3 \ldots 2^l - 1$ from c_1 and it can be seen that the closest request in this set of branches to the right of x is not further away from x than c_1. ∎

We now compute the length of the greedy walk. It traverses the trunk $k+1$ times and the the branches at least once each. Thus, the length of the greedy walk is $L^A \geq (k+1)w + w_B$ where w_B is the sum of the weights of all the branches. The Euler tour traverses each edge of the tree at most twice, thus $L^* \leq 2(w + w_B)$.

Now, we upper-bound w_B. The number of branches in B_i is given by the following Lemma.

Lemma 10 *We have* $|B_{k-i}| < 2\log^i w$.

Proof: By induction: $|B_k| = 1$, and $|B_{k-1}| = \log w + 1$. With the induction assumption we have $|B_{k-i+1}| < 2\log^{i-1} w$. The maximum distance d between two branches of $|B_{k-i+1}|$ is less than w, therefore $|B_{k-i}| < 2\log^{i-1} w \cdot \log d < 2\log^{i-1} w \cdot \log w = 2\log^i w$. ∎

Thus, w_B is bounded by,

$$w_B = \sum_{i=0}^{k} |B_{k-i}| \cdot (k-i) < 2\sum_{i=0}^{k} (k-i) \cdot \log^i w < 2\frac{\log^{k+1} w}{(\log w - 1)^2}$$

We use $k = \log w / \log\log w$ and get $w_B < 2w$. Thus,

$$L^A/L^* \geq \frac{(k+1)w + w_B}{2(w+w_B)} \geq \frac{(k+1)w}{6w} = \Omega(\log w/\log\log w)$$

This concludes the proof of Theorem 7. ■

4 Discussion

4.1 Stretch

While in the general case it may not be possible to find a tree with low stretch (for a ring with n nodes, the stretch of any tree is $\Theta(n)$), in the typical case, one might find a tree with a "good" stretch. In particular, if the network itself is a tree, then we can find a tree with stretch of 1. Finding good trees to execute the Arrow protocol is studied by Peleg and Reshef in [8]. They note that if the adversary (who decides where requests occur) is oblivious, then one can use approximation of metric spaces by tree metrics [1, 2, 3] to choose a tree with an expected overhead of $O(\log n \log\log n)$ for general graphs and $O(\log n)$ expected overhead for constant dimensional Euclidean graphs.

When combined with results from the previous section, this gives us an $O(\log n \log\log n \log r)$ competitive ratio for the Arrow protocol on a general n-node graph with an oblivious adversary, and r concurrent requests (note that the above competitive ratio is not a worst-case ratio, but an expected ratio, the expectation taken over coin flips during the selection of the spanning tree).

4.2 Special Graphs

Some common graphs do not need the extra $\log r$ overhead. If the network itself is a tree, and there are enough concurrent requests, then we can apply a different analysis to strengthen our result.

Theorem 11 *Let G be a tree with constant degree c. Suppose all requests R come from a subtree T of G. Let h denote the height of the subtree, and let $w(e) = 1$ for each edge e in T. Then the cost of the Arrow protocol is bounded by c^h. If the number of concurrent requests is significant, i.e. $r = |R| = \Omega(c^h)$, then the Arrow protocol is asymptotically optimal, that is, $L^A = O(L^*)$.*

Proof: Let e be an edge in T. Denote the number of times the greedy walk traverses edge e by $t(e)$. The distance between an edge e and vertex v is defined to be the distance between v and the adjacent vertex of e that is closest to v. Let edge e be at level l (at a distance of l from the *root*, $0 \le l \le h-1$).

Let u_i be the request that is visited right after the ith traversal of edge e, with $i = 1, \ldots, t(e)$. Note that node u_i with odd (even) index i has $d_T(root, u_i) > (\leq)\ d_T(root, e)$. Moreover, $d_T(u_i, u_{i+2}) \geq d_T(u_i, u_{i+1})$. Since $d_T(u_i, u_{i+1}) = d_T(u_i, e) + 1 + d_T(e, u_{i+1})$, and $d_T(u_i, u_{i+2}) = d_T(u_i, e) + d_T(e, u_{i+2})$, we conclude that

$$\begin{aligned} d_T(u_{i+2}, e) &= d_T(u_i, u_{i+2}) - d_T(u_i, e) \\ &\geq d_T(u_i, u_{i+1}) - d_T(u_i, e) \\ &= d_T(e, u_{i+1}) + 1. \end{aligned}$$

With $d_T(e, u_1) \geq 0$ we have $d_T(e, u_i) \geq i - 1$. Let k be the greatest odd number which is not greater than $t(e)$. In other words, $t(e) \leq k + 1$. Using $d_T(e, u_k) \geq k - 1$ we get $t(e) \leq d_T(e, u_k) + 2$. Let h be the height of the tree. Since the tree has height h, $d_T(e, u_k) \leq h - l - 1$, thus $t(e) \leq h - l + 1$.

Because the tree has constant degree c, we know that the number of edges at level l is bounded by c^l. The greedy walk is bounded by the sum of traversals of all the edges in the tree, that is

$$greedy(T, R) \leq \sum_{e \in T} t(e) \leq \sum_{l=0}^{h-1} c^l (h - l + 1) = O(c^h).$$

Applying Theorem 1 we immediately get $L^A = O(c^h)$. On the other hand, the optimal TSP tour has to visit at least r nodes, and since no two requests are at the same node, we have $L^* \geq r$. The second claim follows. ∎

If the network G is a linked list, a similar analysis yields:

Theorem 12 *If G is a linked list, then $L^A \leq 2L^*$.*

4.3 Other cost measures

We have so far analyzed the sum of latencies (or equivalently, the average latency of a request). Another cost measure is the delay till all the requests have been queued up. In the Arrow protocol, every request takes a simple path on the tree and never waits. Hence the delay until every request has been queued is trivially bounded by the diameter of the tree.

5 Conclusions and Open Problems

In this paper we presented a competitive analysis of the Arrow distributed queuing protocol for the one-shot problem. The key ideas were a *greedy* characterization of the behavior of the protocol under concurrency and a connection to the nearest-neighbor heuristic for the TSP. We also presented a constructive lower bound on the worst case performance of the nearest neighbor heuristic for the TSP on tree metrics.

Open Problems

In this paper, we analyzed the one-shot instance of the Arrow protocol where all the requests start at the same time and yields a competitive ratio of $s \cdot \log r$. The other end of the spectrum is the *sequential* case, where the requests are so far apart in time that the Arrow protocol and an optimal protocol would choose the same queuing order. For the sequential case, the competitive ratio for Arrow is s. This leads to the following natural question: *can we prove a competitive ratio for the general case, which is neither sequential nor one-shot?*

Trees/requests with greedy walks that are super-linear in the number of nodes have a rather peculiar shape. It seems that most "natural" trees/requests have greedy walks that are only linear in the number of nodes. We pose the following question: *what classes of trees/requests have a linear greedy walk?*

Acknowledgments

We thank Eric Ruppert for helpful discussions and comments about the paper, and the PODC referees for useful feedback.

References

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

[2] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, 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 *Proceedings of the 39th IEEE Symposium on the Foundations of Computer Science*, 1998.

[4] M. Demmer and M. Herlihy. The arrow directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing*, Sept. 1998.

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

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

[7] 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.

[8] D. Peleg and E. Reshef. A variant of the arrow distributed directory protocol with low average complexity. In *Proc 26th International Colloquium on Automata Languages and Programming*, July 1999.

[9] D. Rosenkrantz, R. Stearns, and P. Lewis. An analysis of several heuristics for the traveling salesman problem. *SIAM Journal on Computing*, 6(3):563–581, Sept. 1977.