A Data Tracking Scheme for General Networks

Rajmohan Rajaraman [1] *Andréa W. Richa* [2] *Berthold Vöcking* [3] *Gayathri Vuppuluri* [4]

ABSTRACT

Consider an arbitrary distributed network in which large numbers of objects are continuously being created, replicated, and destroyed. A basic problem arising in such an environment is that of organizing a distributed directory service for locating object copies. In this paper, we present a new *data tracking scheme* for locating nearby copies of objects in arbitrary distributed environments.

Our tracking scheme supports efficient accesses to data objects while keeping the local memory overhead low. In particular, our tracking scheme achieves an expected polylog(n)-approximation in the cost of any access operation, for an arbitrary network. The memory overhead incurred by our scheme is $O(\text{polylog}(n))$ times the maximum number of objects stored at any node, with high probability. We also show that our tracking scheme adapts well to dynamic changes in the network.

[1]College of Computer Science, Northeastern University, Boston, MA 02115, `rraj@ccs.neu.edu`. Supported by NSF CAREER award NSF CCR–9983901.
[2]Department of Computer Science and Engineering, Arizona State University, Tempe, AZ 85287–5406, `aricha@asu.edu`. Supported in part by NSF CAREER Award CCR–9985284 and NSF Grant CCR–9900304.
[3]Max-Planck-Institut für Informatik, Saarbrücken, Germany. Supported in part by the IST Programme of the EU under contrac number IST-1999-14186 (ALCOM-FT)
[4]Compaq Corporation, San Jose, CA, `Gayathri.Vuppuluri@compaq.com`. This work was done while the author was a graduate student at Arizona State University, supported in part by NSF CAREER Award CCR–9985284.

1. INTRODUCTION

Replication is a powerful tool in the design of scalable high-performance distributed systems. For example, the scalability problem that arises when a large number of clients simultaneously access a single object (the "hot spot" problem) can be addressed by creating several copies of the object and then distributing the load among these copies. As another example, the latency associated with accessing an object at some distant node of a large network can often be reduced by caching. Indeed, large-scale replication and cooperative caching are central themes that underlie the emergence of the paradigms of content delivery networks and peer-to-peer networking. A basic problem arising in such replicated data environments is that of organizing a distributed directory service for locating object copies. In this paper, we present a new *data tracking scheme* for locating nearby copies of objects in arbitrary distributed environments. Our tracking scheme describes the control structures that need to be stored at the network nodes, defines protocols for locating nearby copies of data objects, and protocols for updating our control structures in case of insertions and deletions of copies. We also consider the adaptability of our scheme as nodes join and leave the system.

We represent the network by a collection V of n nodes with a single communication cost function that takes into account the combined effect of network parameters such as congestion, edge delays, edge capacities, distance and buffer space. The *cost of communication* is defined by a function $c : V^2 \rightarrow \mathbf{R}^+$. For any two nodes u and v in V, $c(u, v)$ is the cost of transmitting a unit size message from node u to node v. We assume that c is a *metric* — that is, it is symmetric and satisfies the triangle inequality. In the remainder of this paper, we will address the proximity of two nodes u and v — e.g., u is *close* to v, the *distance* between u and v, or u is *near* to v — always with respect to the cost of communication $c(u, v)$ between these nodes.

An important measure of the efficiency of a data tracking scheme is the *stretch factor*, which compares the cost incurred by the tracking scheme to the optimal communication cost. We consider three operations: *access*, whereby a node fetches a copy of a particular object, *insert*, whereby a new copy of an object is added to the system, and *delete*, whereby an exsiting copy of an object is removed from the system. The stretch factor of an access, insert or delete operation at a node u for an object A is the ratio of the actual communication cost incurred by the data tracking system in performing the operation to $c(u,v)$, where $v \neq u$ is the nearest node to u that holds a copy of A; if there is no such v, then we set the stretch factor to be the ratio of the cost incurred to $diam(G)$, where $diam(G) = \max_{u,v} c(u,v)$ is the diameter of the network.

A challenge in designing efficient tracking schemes is to achieve a low stretch factor while maintaining small control structures at the network nodes. Consider a naive approach that optimizes the stretch factor for the read operation by storing at each node the location of the closest copy of each object in the network. The control memory required at each node is prohibitive since it is proportional to the total number of objects in the system. Furthermore, when any object copy is inserted or deleted, a large number of nodes (possibly, all) need to be informed. Clearly, the *memory overhead* required at each node is also an important performance metric of a data tracking scheme. Let $\mathcal{S}$ denote the maximum number of objects that may be stored on any individual node. We relate the amount of memory required by our data tracking scheme to $\mathcal{S}$. Formally, we measure the memory overhead at each node by the fraction of local memory that is used for control structures.

A final performance metric that we consider is how the scheme adapts to dynamic changes in the network under the assumption that the cost metric does not change. We adopt the following model for this study. We assume that the set V of nodes and the cost function are fixed; however, individual nodes may *join* or *leave* the data management system. When a node leaves the system, the functionality provided by the node needs to be taken over by the rest of the network. Similarly, in order to achieve scalability, when new nodes join the system, a portion of the data structure needs to be distributed among the new nodes. We evaluate the *adaptability* of a tracking scheme by the number of nodes that are updated when a node joins or leaves the system.

1.1 Our contributions

Our main contribution in this work is the development of the first data tracking scheme for arbitrary networks, that simultaneously achieves polylogarithmic approximations in stretch factors for access, insert and delete operations, as well as for local memory overhead per node. Our data tracking scheme is based on a randomized hierarchical decomposition technique of Bartal [8], that partitions the network into disjoint clusters at various degrees of locality. The protocol for accessing an object in our tracking scheme is to search for the object level by level, from the smallest clusters to the largest, until an object copy is found (if it exists).

A challenge is then to provide an efficient mechanism for searching within a cluster that keeps the memory overhead low and adapts quickly to changes in the network. For this purpose, we embed a de Bruijn graph [17] of appropriate size into each cluster. We use the embedding to guide access requests within the cluster via hashed pointers to relevant object copies. Since the diameter of a 2^k-node de Bruijn graph is k, the search process within any cluster is efficient.

Another important feature of our tracking scheme is that it needs to store only a small number of *signposts* (pointers) for every data object. Typically, these signposts are tuples of object names (object IDs) and node addresses. We assume that both the object IDs and the node addresses have a unique integer representation that can be stored in a constant number of words. Furthermore, since the in-degree and out-degree of each vertex in a de Bruijn graph is 2, the embedding in each cluster only requires a constant amount of words to be stored at each node. We now summarize the main properties of our data tracking scheme.

- The stretch factor for any read operation is $O(\log^3 n)$.
- The expected stretch factor for any insert or delete operation is $O(\log^3 n)$. (The worst case cost for these operations is $O(diam(G)\log^2 n)$.)
- The local memory requirement is
$$O\left(\mathcal{S}\log(\min\{diam(G), n\})(\log n + \log \mathcal{S}) + \log(\min\{diam(G), n\})\log^2 n\right)$$
words, w.h.p.[1]. Assuming $\mathcal{S}$ is polynomially bounded in n, our upper bound on the local memory requirement simplifies to $O(\mathcal{S}\log^2 n + \log^3 n)$.
- The amortized adaptability of our tracking scheme is $O(\log^2 n)$.

The logarithmic factors in the stretch are partially derived from the properties of a network decomposition of Bartal [9], as we will see in Sections 3 and 4. Utilizing improved clustering techniques may result in better stretch factors. For example, for planar graphs one can divide the bounds for the stretch by a $\log\log n$ factor by employing a clustering algorithm of [15]. Furthermore, increasing the bound on the memory requirement by only an additive $O(n^\epsilon)$ term, for constant $\epsilon > 0$, reduces the bounds on the stretch by a factor of $\Theta(\log n/\log\log n)$. Finally, if one only wants a guarantee for the *expected* stretch for read operations, then one can drop another $\log n$ factor in all bounds for stretch and memory requirement (a simplified such scheme, which only provides guarantees on the expected cost of a read operation, is presented in Section 3).

1.2 Related Work

The clustering and decomposition techniques of Bartal [8, 9, 15] build on the seminal work of Awerbuch and Peleg [6] (see also [4]), who provide the first low-diameter hierarchical decomposition for arbitrary networks. These clustering techniques have found several applications in distributed networks, and network algorithms such as maintaining routing tables [5], distributed data management [2, 10, 3], tracking of mobile users [7], network design [22], and locating Internet servers [13].

Closely related to our work is the study of distributed paging [3, 23], which addresses a more general online adversarial version of the problem that we consider in this pa-

[1]We use the abbreviation "w.h.p." throughout the paper to mean "with high probability" or, more precisely, "with probability $1 - n^{-c}$, where n is the number of nodes in the network and c is a constant that can be set arbitrarily large by appropriately adjusting other constants defined within the relevant context."

per. The distributed algorithm of [3] achieves polylog(n)-competitiveness in terms of access cost. Their study does not address the overhead due to control information, however, and direct extension of their results to our problem may require local memory proportional to the number of objects.

The randomized tracking scheme of Plaxton, Rajaraman, and Richa [20] addresses many of the same concerns that we consider in this paper. Their protocol, which has also been implemented as part of a large-scale persistent object repository named Oceanstore [16], achieves a constant-factor approximation in expected access cost for a restricted class of communication cost functions motivated by hierarchical networks. Ours, in contrast, achieves a polylog(n)-approximation for arbitrary networks.

The idea of partioning the network into clusters and allocating pointers (and copies) inside the clusters using hash functions was introduced by Maggs et al. in [18]. The data manangement schemes presented in [18], however, assume unbounded memory capacities and are restricted to structured networks like meshes or hierarchical networks. In [19], these schemes are generalized to broader classes of networks and, furthermore, local memory constraints are incorporated. These memory constraints, however, are only w.r.t. the stored copies but not the pointers to these copies, which in turn is our main concern in this paper. Other work on data management for restricted network models can be found, e.g., in [1] and [14].

2. TECHNICAL OVERVIEW

Our scheme is based on a *hierarchical clustering* $H = H(G)$ of a network $G = (V, E)$. We break the network into disjoint clusters (i.e., subsets of nodes) with smaller diameter[2]. These clusters are partitioned recursively until we reach single nodes. In particular, we demand that the diameter of a child cluster is at most half the diameter of the parent cluster.

The clustering defines a decomposition tree $T(H)$ whose nodes represent the clusters of H. In particular, the root of the tree represents the cluster V, and the children of a cluster C represent the clusters into which C is partitioned according to H. Also, every leaf of the decomposition tree corresponds to a cluster containing a single node, and therefore represents a distinct node of the graph G. The *length of an edge* in $T(H)$ connecting a parent cluster C with a child cluster C' is defined by $diam(C)$, where $diam(C)$ denotes the (weak) diameter of C.

We will use randomly generated cluster constructions: The cluster hierarchy H is chosen at random from a distribution $\mathcal{H} = \mathcal{H}(G)$ of hierarchical clusterings on G. The performance of our data tracking scheme depends on the quality of the hierarchical clusterings obtained by the randomized construction scheme. The most important quality measure is the *stretch factor* of H, denoted by $s(\mathcal{H})$, which is equal to the maximum *stretch factor* $s_{u,v}(\mathcal{H})$ over all pairs of nodes $u, v \in V$, where

$$s_{u,v}(\mathcal{H}) = \mathrm{Ex}\left[\frac{dist_{T(H)}(u,v)}{dist_G(u,v)}\right] \qquad (1)$$

with $dist_G(u,v)$ and $dist_{T(H)}(u,v)$ denoting the cost of communication between nodes u and v w.r.t. G (given by the cost function c) and as defined by the clustering decomposition tree $T(H)$, respectively (Observe that $dist_{T(H)(u,v)}$ is a random variable w.r.t. the random choice of H from $\mathcal{H}$.). Another quality measure is the *depth* $d(\mathcal{H})$ of the clustering scheme, which we define to be the maximum height of the clustering tree $T(H)$ over all H in $\mathcal{H}$.

In Sections 3 and 4, we will prove the following theorem, which relates the performance of our data tracking scheme to the quality of the randomized clustering scheme. (Recall that n denotes the number of nodes and $\mathcal{S}$ denotes the maximum number of objects that may be held by a single node.)

THEOREM 1. *Given a graph G with clustering scheme $\mathcal{H} = \mathcal{H}(G)$, there exists a randomized data tracking algorithm with (deterministic) stretch factor $O(s(\mathcal{H})\log_k n \log n)$ for access requests, expected stretch factor $O(s(\mathcal{H})\log_k n \log n)$ for insert and delete requests and memory overhead of*

$$O\left(\mathcal{S}d(\mathcal{H})(\log n + \log \mathcal{S}) + d(\mathcal{H})(k + \log n)\log n\right) \ ,$$

words at each node, w.h.p, for every $2 \le k \le n$.

It remains to describe how the results presented in Section 1 can be derived from Theorem 1. In [8], Bartal presents a probabilistic approximation of metric spaces by so-called hierarchical well separated trees (HSTs). In fact, the construction of these trees is based on a hierarchical partitioning scheme with small stretch and depth. Meanwhile the original results of Bartal have been improved. The currently best known bounds are as follows.

- For general graphs, $s(\mathcal{H})$ equals $\log n \log\log n$ and $d(\mathcal{H})$ equals $\min\{diam(G), \log n\}$ [9].
- For planar graphs, $s(\mathcal{H})$ equals $\log n$ and $d(\mathcal{H})$ equals $\min\{diam(G), \log n\}$ [15].

Combining these bounds with Theorem 1 (for $k = \log n$) yields the results stated in Section 1.1.

In Section 3, we will show how to obtain an upper bound on the expected stretch factor for all operations and on the memory overhead. Then, in Section 4, we will build on the results proved in Section 3, showing how to achieve a deterministic stretch for the access operation if we use $\log n$ copies of the data structure defined in Section 3. Finally, in Section 5 we will investigate the adaptability of our scheme. We conclude with some directions for future work in Section 6.

3. MINIMIZING EXPECTED STRETCH

In this section, we will show a slightly weaker version of Theorem 1. The following lemma bounds only the expected stretch for access operations.

[2]We consider *weak diameters*. The *(weak) diameter* of a cluster C is the maximum cost of communication *in* G between any pair of nodes in C.

MAIN LEMMA 2. *For every graph G with randomized clustering scheme $\mathcal{H} = \mathcal{H}(G)$ there exists a data tracking algorithm with expected stretch $O(s(\mathcal{H}) \log_k n)$ and local memory requirement*

$$O\left(d(\mathcal{H})\left(\mathcal{S}\left(1+\frac{\log \mathcal{S}}{\log n}\right)+\log n+k\right)\right) ,$$

words, w.h.p., for every $2 \le k \le n$.

In Section 3.1, we will first present a simple data tracking scheme that achieves small expected stretch but stores all information regarding the state of all copies in a cluster on a single node, which we refer to as a *cluster leader.* In Sections 3.2 and 3.3, we will show how to reduce the local memory requirement by embedding de Bruijn graphs into the clusters.

3.1 The cluster leaders

Let us assume that each cluster has a special node that holds all relevant information about copies in the cluster. For a cluster C, this node is the cluster leader $L(C)$. We maintain the following invariant.

INVARIANT 3. *Suppose C' is the child cluster of a cluster C. For every data object A, $L(C)$ holds a* signpost *for A pointing to $L(C')$ iff C' holds a copy of A.*

Using this invariant, one can always find a close copy by following the shortest path in the decomposition tree $T(H)$. Recall that a node $u \in V$ corresponds to a leave in $T(H)$. If u searches for an object A then it simply can send a message upward in the decomposition tree until it reaches a signpost for A, that is,

- the message follows the chain of cluster leaders representing the clusters on the path upward in the decomposition tree,
- the message is stopped as soon as it reaches a cluster leader $L(C)$ of a cluster C holding a signpost for A, and then
- the message follows the chain of signposts downwards until it reaches a copy of A on a node v.

The above path is called the search path of u for A. We denote its length by $\ell(u, A)$. Clearly, if u issues an access request to A, then one only has to perform a search to a copy and return the absolute address of node v. In case of an insert of a copy on u, in each cluster C one follows the search path of u for A and, at every cluster C on this path, one follows a path to $L(C)$, adding a signpost for A at $L(C)$. Similarly, in case of a delete of a copy on u, one follows the search path of u for A and, for every cluster C on this path, one follows a path to $L(C)$, removing the signpost for A at $L(C)$. Thus, the cost for access, insert and delete operations are bounded above by $O(\ell(u, A))$.

Next we give an upper bound on $\ell(u, A)$. For notational convenience, we let $\text{dist}_G(u, A)$ (resp., $\text{dist}_{T(H)}(u, A)$) denote the distance in G (resp., $T(H)$) between u and the closest copy of A on another node.

LEMMA 4. $E[\ell(u, A)] \le 2s(\mathcal{H}) \cdot \text{dist}_G(u, A)$.

PROOF. Let v_G and $v_{T(H)}$ denote the closest node wrt G and $T(H)$, resp., holding a copy of A. Observe that possibly, $v \ne v_{T(H)} \ne v_G$. Invariant 3, however, yields that we always find a copy in the smallest cluster containing u and a copy of A. Since the costs for edges in $T(H)$ decrease geometrically by a factor of two from the root to the leaves, we obtain

$$\ell(u, A) = \text{dist}_{T(H)}(u, v) \le 2\text{dist}_{T(H)}(u, v_{T(H)}).$$

We now calculate the expected value of $\text{dist}_{T(H)}(u, v_{T(H)})$.

$$\begin{aligned} E[\text{dist}_{T(H)}(u, v_{T(H)})] &\le s(\mathcal{H}) \cdot \text{dist}_G(u, v_{T(H)}) \\ &\le s(\mathcal{H}) \cdot \text{dist}_G(u, v_G) \\ &= s(\mathcal{H}) \cdot \text{dist}_G(u, A), \end{aligned}$$

which yields the lemma. □

A naive implementation of Invariant 3 requires that a cluster leader of a cluster with Δ children has to store up to Δ signposts per cluster. We conclude this section by showing that one can redistribute the signposts in such a way that the memory requirement per cluster leader is independent from Δ.

LEMMA 5. *Using cluster leaders, every access, insert, or delete operation of a node u wrt an object A can be performed at cost $O(\ell(u, A))$ storing only $O(1)$ signposts for A on the leader of those clusters that contain a copy of A.*

PROOF. Let $C_1, \dots, C_\delta$ denote the child clusters of $C = C_0$ that contain a copy of A. We connect these clusters in a doubly linked list so that $L(C_i)$ holds pointers to $L(C_{i+1})$ and $L(C_{i-1})$, for $0 \le i \le \delta$. In this way, one can efficiently search for a copy in C by following the pointer to C_1. If a first copy of A is inserted in a different child cluster, then we add the leader of this cluster to the head of the linked list. If the last copy of A is deleted from a child cluster, then this cluster is removed from the linked list. If this leaves an empty linked list, then C has no copies of A, and $L(C)$ removes itself from the linked list at the next level of the hierarchy. Thus, insertions and deletions of copies are implemented using standard operations for doubly lists requiring only to change two pointers in the list. Each change of a pointer costs $O(diam(C))$. Thus, the asymptotic cost for insertions and deletions of copies does not change.

□

3.2 Distributing the cluster leadership

An obvious drawback of the cluster leader concept is that the leader node needs to store signposts to all copies in a cluster (eventually signposts to all copies in the whole network). To overcome this problem one might define different leader nodes for different objects using a hash function that distributes the signposts evenly among the nodes in a cluster. A naive implementation of this concept, however, assumes that every node is known by every other node in the cluster. For example, a typical implementation of a hash function requires that the nodes in a cluster are numbered in a consecutive fashion. However, even if the nodes in the network are numbered from 0 to $n-1$, the locality conditions of the clusters can produce arbitrary subsets of these numbers within a given cluster. Therefore, we number the nodes in different clusters independently. In this way, we can compute a hash function that maps signposts to nodes. However, we do not want to store gigantic tables translating the labels for all nodes in all clusters into physical addresses. Instead we locate the pseudo-randomly distributed signposts by following shortest paths in an embedded de Bruijn graph. In this way, we ensure that every node has to store only its own label and the labels of a few other nodes in each of the $d(\mathcal{H})$ clusters in which it is contained. We now describe the hashing scheme and the embedding in more detail.

Hash function.

We assign *keys* to the objects. For an object $A \in \mathcal{A}$, let $key(A)$ denote the key of object A. Keys are not unique, we choose them from the set $[P]^3$ using a hash function, where $P \geq |\mathcal{A}|$ is a prime number. The hash function is chosen as follows. Following Carter and Wegmann [11], we draw a polynomial f from a class of integer polynomials $\mathcal{F}$ of degree $q = O(\log(\mathcal{S}n))$. (Observe that the representation of f requires only q words.) We define $key(A) = f(int(A))$, where $int(A)$ is a unique integer representation of A in $\mathbf{Z}_P$. This polynomial hashing scheme guarantees *q-wise independence.* In particular, we can conclude the following lemma.

LEMMA 6 (CARTER AND WEGMANN [11]). *Let $M \leq P$ and $m \in [M]$ be two integers. For every collection of q distinct objects $A_1, \ldots, A_q$,*

$$\Pr\left[(key(A_i) \bmod M) = m \text{ for } 1 \leq i \leq q\right] \leq \left(\frac{2}{M}\right)^q .$$

Now, for each cluster C, we number the nodes in C from 0 to $|C-1|$. The signposts of an object A are stored in the node with label $home_C(A) = key(A) \mod |C|$.

Embedding de Bruijn graphs.

In order to avoid large tables that translate the virtual node labels within the clusters into physical addresses, we embed a $\lceil \log |C| \rceil$-dimensional de Bruijn graph into each cluster C. For convenience, we let $d = \lceil \log |C| \rceil$ in the following discussion. The d-dimensional de Bruijn graph consists of 2^d vertices whose labels are d-ary binary strings that can be identified with integers from $0 \ldots d-1$. The nodes in the cluster C are assigned labels from $[|C|]$. Let $\lambda_C(u)$ denote the label of u in C. Any de Bruijn vertex with integer label $\ell \in [|C|]$ is hosted by the cluster node u with label $\lambda_C(u) = \ell$. Any de Bruijn vertex with integer label $\ell \geq |C|$ is hosted by the cluster node u whose label $\lambda_C(u)$ is identical to ℓ in binary representation, except that the most significant bit in $\lambda_C(u)$ is 0 instead of 1. Observe that each cluster node hosts either one or two de Bruijn vertices.

In the de Bruijn graph, there is a directed edge from each vertex with label $u_1 u_2 \ldots u_d$ to the vertices with labels $u_2 \ldots u_d 0$ and $u_2 \ldots u_d 1$. The diameter of this directed graph is $\log C$ and there is a unique shortest path between every pair of nodes that can be computed easily. (For more details about the de Bruin graphs see, e.g., [17].) If a node v in cluster C needs to send a message to node $home_C(A)$, for some object A, then this message follows the edges on the shortest path from v to $home_C(A)$ in the de Bruijn network. In this way, each node on this path only needs to know the physical address of the next node on the path. This information can be obtained easily if every node stores the physical addresses of the nodes incident on its outgoing edges. We refer to this table as the *neighbor table.* Since the out-degree of each vertex in the de Bruijn graph is 2 and at most 2 vertices are emulated by any cluster node, the neighbor table at each node is constant size.

The price for routing messages along shortest paths in the de Bruijn network is that the search within a cluster has cost $O(diam(C) \log |C|)$ rather than $O(diam(C))$ because it visits up to $\lceil \log |C| \rceil - 1$ intermediate nodes. Clearly, one can save cost by storing a larger neighborhood in the neighbor table. This yields the following tradeoff. If a node memorizes a neighborhood of size $k \geq 2$ for each of its at most two de Bruijn vertices of cluster C, then a search in cluster C has cost $O(diam(C) \log_k |C|)$. Consequently, adapting the bound on the cost of access, insert, and delete operations in Lemma 5 to the embedding yields the following result.

LEMMA 7. *For every $k \geq 2$,*

- *every access, insert, or delete operation of a node u wrt an object A can be performed at cost $O(\ell(u, A) \log_k n)$*
- *for every cluster C, each node $v \in C$ needs to memorize $O(k)$ words to store the de Bruijn neighborhood, and*
- *for every cluster C, each node $v \in C$ needs to hold $O(1)$ signposts for every object A if $home_C(A) = v$ and cluster C contains copies of A.*

3.3 Analysis of memory requirement

It remains to count all labels, addresses and signposts over all clusters that need to be stored in the local memory modules of the nodes.

LEMMA 8. *Let $\mathcal{S}$ denote the maximum number of objects that can be stored in the main memory of any node. Let k denote the number of memorized de Bruijn neighbors. Then the local memory requirement is*

$$O\left(d(\mathcal{H})\left(\mathcal{S}\left(1 + \frac{\log \mathcal{S}}{\log n}\right) + \log n + k\right)\right),$$

words, w.h.p.

PROOF. A node is contained in $d(\mathcal{H})$ clusters. We will show, for every cluster C, each node $v \in C$ holds at most $O((\mathcal{S} + \log n)(\log n)^{-1} \log(n\mathcal{S}))$ signposts, w.h.p. Lemma 7 shows that the additional number of words that need to be stored for the de Bruijn neighborhood is $O(k)$ per cluster.

[3] For any nonnegative integer x, $[x]$ denotes the set $\{0, 1, \ldots, x-1\}$.

Furthermore, we will need an $q = O(\log(n\mathcal{S}))$-wise independent hash function in order to show the upper bound on the number of signposts. For the representation of this hash function we require $O(\log(n\mathcal{S}))$ words of memory in every node. Putting altogether, we obtain an upper bound on the local memory requirement of

$$O\left(d(\mathcal{H})\left((\mathcal{S}+\log n)\frac{\log(n\mathcal{S})}{\log n}+k\right)+\log(n\mathcal{S})\right),$$

which after simplification corresponds to the bound in the lemma.

We now place an upper bound on the number of signposts stored at a node. Fix a cluster C. We show that each node v in C holds at most $O((\mathcal{S}+\log n)(\log n)^{-1}\log(n\mathcal{S}))$ signposts, with probability $1-n^{-c}$, for any constant $c>0$. Recall that only the node $home_C(A) = key(A) \mod |C|$ may hold a signpost directed to a copy of A where $key(A)$ is defined by a q-wise independent hash function.

Let R denote the set of objects with at least one copy in cluster C. We partition R into $\kappa = \lceil \mathcal{S}/\log n \rceil$ groups $R_1, \ldots, R_\kappa$ each of which having size at most

$$2|R|\frac{\log n}{\mathcal{S}} \le 2\log n|C| \ .$$

(The last inequality holds because $|R| \le \mathcal{S}|C|$ as each node in C can store at most $\mathcal{S}$ objects.) For a node $v \in C$, let $r_i(v)$ $(1 \le i \le \kappa)$ denote the number of signposts for objects from R_i that are stored on v.

Our hash function aims to distribute the signposts evenly among the nodes in the cluster. In fact, applying Lemma 6 yields

$$\begin{aligned}\Pr[r_i(v) \ge q] &\le \binom{2\log n|C|}{q}\left(\frac{2}{|C|}\right)^q \\ &\le \left(\frac{4e\log n}{q}\right)^q \\ &\le \mathcal{S}^{-1}n^{-c},\end{aligned}$$

provided $q = c_1\log(n\mathcal{S})$ with $c_1 > 0$ denoting a suitable constant. Now let $r(v)$ denote the number of signposts on v for all objects in $R = \bigcup_{i=1}^{\kappa} R(i)$. Then

$$\Pr[r(v) \ge \kappa q] \le \kappa\mathcal{S}^{-1}n^{-c} \le n^{-c}.$$

Finally observe that

$$\kappa q = \left\lceil \frac{\mathcal{S}}{\log n} \right\rceil c_1\log(n\mathcal{S}) \;=\; O\left(\frac{\mathcal{S}+\log n}{\log n}\log(n\mathcal{S})\right).$$

This completes the proof of Lemma 8. $\square$

Combining the bounds in Lemma 4, Lemma 7, and Lemma 8 yields Main Lemma 2 stated at the beginning of this section.

4. DETERMINISTIC STRETCH

The probability for the bound on the expected stretch in the Main Lemma in Section 3 is w.r.t. the randomized construction of the hierarchical clustering of Bartal [8, 9]. This means that there are possibly some allocations of copies to nodes in which accesses issued by particular nodes are always very expensive. This may be acceptable for insert or delete requests for which one typically aims to minimize the overall work load, but it is not acceptable for access requests which typically occur much more frequently, and for which one aims to minimize the latency for any particular request. The following lemma addresses this problem.

LEMMA 9. *Using $O(\log n)$ copies of the data tracking scheme presented in Section 3, one can ensure a deterministic stretch factor of $O(s(\mathcal{H})\log_k n\log n)$ for an access operation, and an expected stretch factor of $O(s(\mathcal{H})\log_k n\log n)$ for the insert and delete operations.*

PROOF. Consider a randomized clustering scheme $\mathcal{H}$ generating a hierarchical clustering $H(G)$ with stretch $s(\mathcal{H})$. Applying the Markov inequality to $\mathrm{Ex}[\mathrm{dist}_{T(H)}(u,v)] \le s(\mathcal{H})$ yields

$$\Pr[dist_{T(H)}(u,v) \ge 2s(\mathcal{H})] \quad\le\quad \frac{1}{2}\ ,$$

for every pair of nodes $u, v \in V$. Now suppose we use $\mathcal{H}$ for generating $r = 2\log n$ independent hierarchical clusterings $H_1, \ldots, H_r$. Then

$$\begin{aligned}\Pr[\exists u,v \in V,\, \forall i \in \{1,\ldots,r\} : dist_{T(H_i)}(u,v) \ge 2s(\mathcal{H})] \quad (2)\\ \le \frac{n(n-1)}{2}\left(\frac{1}{2}\right)^r \le \frac{1}{2}. \quad (3)\end{aligned}$$

Initially, our data tracking scheme repeatedly generates r-tuples of hierarchical partitions terminating with the first tuple $H^* = (H_1^*, \ldots, H_r^*)$ satisfying

$$\max_{u,v\in V}\min_{1\le i\le r} dist_{H_i^*}(u,v) \le 2s(\mathcal{H}).$$

Equation 2 shows that this process terminates after generating only $O(\log n)$ tuples, w.h.p.

We implement r versions of our data tracking scheme, the ith version is based on partitioning H_i^*. Insert and delete operations are always executed in all versions. The expected cost for these operations increases by at most a factor of $2r = O(\log n)$ since equation 2 implies

$$\sum_{1\le i\le r}\mathrm{Ex}[dist_{H_i}(u,v)] \quad\le\quad 2rs(\mathcal{H})$$

because the randomized process generating the partitions chooses effectively an s-tuple from a probability distribution $(\mathcal{H}^*)^s$ in which each s-tuple has at most twice the probability as in $\mathcal{H}^s$. Thus, the expected stretch for insert and delete operations increases from $O((\mathcal{H})\log_k n)$ to $O(s(\mathcal{H})\log_k n\log n)$.

In order to perform a search procedure for an object A initiated at a node u, we alternate in round robin fashion among the r versions of the data tracking scheme. In a first round, for every version, we search for copies of A in the smallest clusters containing u. If we do not find a copy in of A in one of these clusters then, in the next iteration, we inspect clusters of the next higher level of all the clustering hierarchies. (A more practical implementation may inspect all clusters of the same level of all versions in parallel.) This way, an access operation always locates a copy of A at distance $2\mathrm{dist}_G(u,A)s(\mathcal{H})$. Thus, the cost for performing a access operation is $2r\mathrm{dist}_G(u,A)s(\mathcal{H})\log_k n$ and, hence, the stretch factor for a access operation is $O(s(\mathcal{H}) \log_k n \log n)$. This completes the proof of Lemma 9. $\square$

The modified data tracking scheme in the proof of Lemma 9 incurs an extra $O(\log n)$ factor in the memory overhead at each node: This fact combined with Lemma 9 concludes the proof of Theorem 1.

5. ADAPTABILITY

An important aspect of our tracking scheme is its adaptability to changes in network conditions. We consider a scenario in which the nodes of the network may leave or join the system over time. We define the *adaptability* of a tracking scheme to be the number of nodes that have to be updated when a node joins or leaves. We show that the amortized adaptability of our tracking scheme is $O(\log^2 n)$.

We first address the case when a node u leaves the system. Consider a cluster C to which u belongs. If $\lambda_C(u)$ is $|C|-1$, then there are two cases. If $|C|-1$ is not a power of 2, then the label $\lambda_C(u)$ is emulated by the node u' that has a label identical to that of u except in bit position 0. The neighbor table at u' (that lists the neighbors according to the de Bruijn embedding) is updated to account for the new neighbors associated with the label. The neighbor table at the nodes that have de Bruijn edges to u need to update their neighbor table as well. Furthermore, the list of signposts at u' is also updated to include the signposts at u. Thus, the total number of nodes that are updated is $O(1)$. If $|C|-1$ is a power of 2, then each node other than u has two labels. On the removal of u, the dimension of the embedded de Bruijn graph decreases by 1. Each node now maintains exactly one of its labels and merges the signpost lists associated with the two labels. Thus, all of the $|C|$ nodes are updated. We finally consider the case in which $\lambda_C(u)$ is less than $|C|-1$. In this case, we set the label of the node with current label $|C|-1$ to $\lambda_C(u)$, and then repeat the updates associated with the removal of the node with label $|C|-1$. Again, the number of nodes updated is $O(1)$. Since the adaptability is $O(1)$ whenever $|C|-1$ is not a power of 2, and $O(|C|)$ otherwise, a simple amortization argument shows that the amortized adaptability in any sequence of node departures is $O(1)$.

When a node u joins a cluster C, then it is assigned a new label $|C|$. There are two cases. If $|C|+1$ is not a power of 2, then the node u' that was previously emulating the label $|C|$ and the nodes that have de Bruijn edges to u' need to be updated. Thus, the number of updates is $O(1)$. If $|C|+1$ is a power of 2, then prior to the addition of u, each node had exactly 1 label. But after the addition of u, each node other than u needs to emulate two labels. The dimension of the embedded de Bruijn graph also increases by 1. Consequently, the number of nodes updated is $|C|$. Since this cost can be amortized against earlier node additions, we obtain in a sequence of node additions, the amortized adaptability within a cluster is $O1)$.

The above adaptation mechanism does not achieve amortized $O(1)$ adaptability within a cluster for a sequence that contains nodes *both* joining and leaving the system. We achieve this bound by allowing the size of the de Bruijn graph to be up to four times the size of the cluster. This increases the number of labels emulated by a node by at most a factor of 2. When a node leaves, the relabeling of the nodes is done only when the ratio of the number of labels to the size of the cluster is less than 4. Using standard amortized analysis, we show that the amortized number of nodes updated within a cluster for any sequence of network changes is $O(1)$. Since each node belongs to $O(\log^2 n)$ clusters, the amortized adaptability is $O(\log^2 n)$.

6. FUTURE WORK

It would be interesting to devise low-stretch clustering hierarchies that could adapt well to a highly dynamic network environment. Such a hierarchy, when combined with our data tracking scheme, would provide a tracking scheme that could be used, for example, in mobile network scenarios or in networks whose data traffic pattern tends to change often (also changing the costs of communication between nodes).

Another extension would be to develop data tracking schemes for objects that may appear in different representation formats in the network (e.g., image and video files is a distributed multimedia network). A node may recognize only a few representation formats among the many formats available for an object. Some nodes in the network may be able to perform a format conversion, at some specified cost. Therefore the cost of a read operation now depends not only on the communication costs between pair of nodes, but also on the conversion costs at the nodes.

7. REFERENCES

[1] T. E. Anderson, M. D. Dahlin, J. N. Neefe, D. A. Patterson, D. S. Rosselli, and R. Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, 1995.

[2] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 164–173, May 1993.

[3] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 574–583, January 1996.

[4] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-linear cost sequential and distributed constructions. In *Proceedings of the Thirty-Fourth Annual Symposium on Foundations of Computer Science*, pages 638–647, 1993.

[5] B. Awerbuch and D. Peleg. Routing with polynomial communication space tradeoff. *SIAM Journal on Discrete Mathematics*, 5:151–162, 1990.

[6] B. Awerbuch and D. Peleg. Sparse partitions. In *Proceedings of the Thirty-First Annual IEEE Symposium on Foundations of Computer Science*, pages 503–513, October 1990.

[7] B. Awerbuch and D. Peleg. Online tracking of mobile users. *Journal of the ACM*, 42(5):1021–1058, September 1995.

[8] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the Thirty-Seventh Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, October 1996.

[9] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 161–168, 1998.

[10] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. *Journal of Computer and System Sciences*, 51:341–358, 1995.

[11] J. L. Carter and M. N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[12] J. D. Guyton and M. F. Schwartz. Locating nearby copies of replicated Internet servers. In *Proceedings of ACM SIGCOMM*, pages 288–298, 1995.

[13] S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L.Zhang, The placement of Internet instrumentation, In *Proceedings of IEEE INFOCOMM*, 2000.

[14] D. Karger, E. Lehman, F. T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1997.

[15] G. Konjevod, R. Ravi, and F. S. Salman. On approximating planar metrics by tree metrics. Submitted for publication in *J. Algorithms*, July 1997.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[17] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays • Trees • Hypercubes.* Morgan Kaufmann, San Mateo, CA, 1992.

[18] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *Proceedings of the Thirty-Eighth Annual Symposium on Foundations of Computer Science*, pages 284–293, October 1997.

[19] F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Caching in networks. In *Proceedings of the Eleventh Symposium on Discrete Algorithms*, pages 430–439, 2000.

[20] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32:241–180, 1999. A preliminary version of this paper appeared in *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 311–320, June 1997.

[21] M. van Steen, F. J. Hauck, and A. S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proceedings of the 1996 Conference on Telecommunications Information Networking Architecture (TINA 96)*, pages 203–212, September 1996.

[22] B. Wu, G. Lancia, V. Bafna, K. Chao, R. Ravi, and C. Tang", A polynomial time approximation scheme for Minimum Routing Cost Spanning Trees In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 21–32, January 1998.

[23] O. Wolfson, S. Jajodia, and Y. Huang An adaptive data replication algorithm *ACM Transactions on Database Systems*, 22:255–314, 1997.