Slicing Distributed Systems

Vincent Gramoli[*], Ymir Vigfusson[†], Ken Birman[†],

Anne-Marie Kermarrec[‡], Robbert van Renesse[†]

[*]EPFL LPD — University of Neuchâtel — vincent.gramoli@epfl.ch

[†]Cornell University — Ithaca, NY — {ymir,ken,rvr}@cornell.edu

[‡]INRIA Rennes — Bretagne Atlantique, — anne-marie.kermarrec@inria.fr

Abstract

Peer-to-peer (P2P) architectures support collaboration to accomplish tasks such as downloading data, VOIP telephone, and cooperative backups. However, end-user systems can be extremely heterogeneous, with heavy-tailed distributions of attributes such as storage space and bandwidth. In systems that ignore heterogeneity, performance suffers, hence most of the popular peer-to-peer applications are forced to classify nodes according to capacity, distinguishing super-peers (which play more active roles) from regular ones (which have limited roles). Similar issues arise in large data centers, where nodes may have widely variable configurations and performance. Our paper solves a generalized classification problem called *slicing*, which involves partitioning the nodes into k subsets using a one-dimensional attribute. Here, we start by arguing that slicing is the most appropriate generalization of existing classification mechanisms. We review prior work on the problem, and introduce our new *Sliver* protocol. Theoretical and experimental evaluations show that Sliver converges more rapidly than alternatives, and its low cost makes it appealing in a wide range of practical settings.

Index Terms

Slicing, Churn, Superpeer, Peer-to-peer.

Technical Areas: Operating Systems and Middleware, Internet Computing and Applications, Peer-to-Peer.

A brief announcement of this article appeared in the proceedings of the 27th Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2008). This work was supported, in part, by NSF, AFRL, AFOSR, INRIA, and Intel Corporation.

I. INTRODUCTION

Peer-to-peer (P2P) protocols have emerged as the technology of choice for purposes such as building VoIP overlays and sharing files or storage. The algorithms discussed here are intended to assist such applications in dealing with heterogeneous resource distributions [22], [25]. For example, VoIP applications such as Skype [24] must avoid routing calls through nodes that are sluggish or have low bandwidth, and file sharing services such as Gnutella [13] and Kazaa [19] employ a two-level structure, in which peers with longer lifetime and greater bandwidth capabilities function as *superpeers*.

Our goal is to solve a generalized classification problem called *slicing*, in which n nodes are grouped into k equally balanced *slices* in a one-dimensional attribute space, such that when the classification algorithm terminates, each node knows the slice index to which it belongs. The resulting classification can be used in different ways by different applications. For example, with $k = 4$, we organize the nodes into quartiles, but an application wishing to treat the top half of nodes as super peers would simply group the top two slices. Although beyond the scope of this paper, our protocol can trivially be extended to slice multiple attributes concurrently (each message would just carry distinct fields for each of the attributes of interest).

Some examples of possible applications include:

- Live streaming applications that need to classify nodes according to their download and upload bandwidths.
- File-sharing services in which nodes having the largest collections of files play distinguished roles.
- Distributed backup services that place large objects on nodes with the most free storage.
- Load-balancers in datacenters that send more work to machines that are less heavily loaded.

We are not the first to study slicing. In [17], the authors describe a communication-efficient parallel sorting algorithm and present node classification as a possible application, but the implicit approach to slicing is sensitive to non-uniform attribute value distributions and churn correlated to attribute values. An accurate slicing algorithm called the *Ranking protocol* was presented in subsequent work [9], but with slow convergence. In larger deployments, membership churn disrupts the underlying system and prevents the algorithm from stabilizing.

These observations motivate us to seek a slicing algorithm that satisfies the following properties:

(i) Efficient and accurate computation of slice indices, as formalized below.
(ii) Rapid convergence to optimal slice indices, with provable guarantees.
(iii) Robustness to membership churn and evolution of underlying attribute values.

The protocol should also be simple, both to facilitate implementation and for ease of analysis.

A. Contributions

The main contribution of this article is now new randomized algorithm, *Sliver* (*Slicing Very Rapidly*), which we compare with two prior solutions to the problem. We also touch briefly on other ways of solving the problem, using parallel sorting algorithms (these turn out to be fast but very sensitive to churn). Sliver is simple, achieves all of our goals, and is very fast. We undertake a rigorous convergence analysis, and an experimental evaluation using real storage and churn traces. Sliver retains its good behavior even in very challenging conditions.

B. Outline

The model of our system, the slicing problem, and preliminary definitions are presented in Section II. Section III compares the slicing problem to the sorting problem, and outlines the limitations of the sorting algorithms. Section IV presents solutions that converge to a sliced network, including Sliver. We analyze this protocol theoretically in Section V and compare its performance to that of other slicing protocols under various settings in Section VI. Finally, Section VII concludes the article.

II. Problem and Model Definition

This section formalizes the model and gives a more precise definition of the slicing problem.

A. System Model

The system consists of n nodes with unique identifiers (e.g., IP addresses); each node knows of a small number of neighbors, and the resulting graph is closed under transitivity. We assume that n is large: solutions that collect $\Omega(n)$ information at any single node are impractical. Time passes in discrete steps starting from time 0.

Each node can leave (or fail by halting) and new nodes can join the system at any time (so-called *churn*), thus the number of nodes is a function of time. A system with no membership

churn is *static*. In this article, we do not differentiate between a failure and a voluntary node departure and we say that the nodes currently in the system are *active*. Let A_t denote the set of active nodes at time t, and let $n_t = |A_t|$ be the number of active nodes at that time ($n_t \leq n$).

At any time t, each node i has an *attribute value* $a_i(t) \in \mathbb{R}$ that represents its capacity in the metric of interest, for example uplink bandwidth. These attribute values can have an arbitrary skewed distribution.

When attribute values remain fixed over time, that is $a_i(t) = a_i(t')$ for all t and t', we refer to i's constant attribute value simply as a_i. Additionally, throughout the paper, we sometimes omit t where it is clear from context.

Every node i keeps an array of records about its neighbors. A record includes the neighbor's identifier i', the last time a message was received from i', the attribute value $a_{i'}$ of i', and optionally the value that i' estimates to be its position. This array, denoted $\mathcal{N}_i$, is called the *view* of node i. To bound the required memory, every node has a view of at most c neighbors where c is a global constant.

B. Definitions

At any time t, we can define a total ordering over the nodes based on their attribute value, with the node identifier used to break ties. Formally, we say node i *precedes* i' at t if and only if $a_i(t) < a_{i'}(t)$, or $a_i(t) = a_{i'}(t)$ and $i < i'$. We refer to this totally ordered sequence as the *attribute sequence*. The attribute-based rank of a node i at time t, denoted by $\alpha_i(t) \in \{1, \ldots, n_t\}$, is defined as the index of a_i in the attribute sequence. We denote by $p_i(t) = \frac{\alpha_i(t)}{n_t}$ the *position* of node i in the system at time t and by $\hat{p}_i(t)$ its *position estimate* at time t. In other words, the position $p_i(t)$ of node i at time t is the index of $a_i(t)$ within the sorted attribute values, normalized to fall within the range $(0, 1]$.

In the remainder of the article, we assume that nodes are sorted according to a single attribute. As mentioned earlier, our protocol has a trivial generalization in which slice multiple attributes concurrently: it suffices to extend messages so that they can carry an array of information, one entry for each attribute (thus, no additional messages are needed). One can also imagine more sophisticated generations involving multiple dimensions. However, the need for brevity precludes an exploration of this topic.

Suppose we partition the attribute sequence at time t into k equally balanced sets. We call

each set a *slice*, and preserve the order within the partition such that the jth slice has a *slice index* j. More formally, the slice with slice index j is the set

$$S_j(t) = \left\{ i \in A_t : \frac{j-1}{k} < p_i(t) \leq \frac{j}{k} \right\}$$

for $1 \leq j \leq k$. Each node belongs to exactly one slice. For example, when $k = 4$ the values of the nodes in the slices correspond to the quartiles of the attribute value distribution. A *slice boundary* refers to a value $\frac{j}{k}$ for some j, delimiting the positions of nodes belonging to the jth slice and nodes belonging to the $(j+1)^{st}$ slice.

Initially, nodes have no global information about the structure or size of the system, or about the attribute values of any other node. We assume that k and c are global knowledge because both can be easily provided to newly joining nodes.

As noted earlier, the model can be easily generalized. In addition to higher-dimensional slicing, interesting options include support for unevenly balanced slice sizes, and protocols that fix the slice size, allowing k to vary. We leave the exploration of these questions for future study.

C. Distributed Slicing

In the *slicing problem* all nodes try to discover the number of the slice to which their attribute value belongs. The correct slice index $o_i(t)$ of slice i at t is the index of the unique slice $S_j(t)$ which contains $a_i(t)$.

Suppose each node i *estimates* its slice index to be $e_i(t)$ at time t. To measure the overall quality of the estimates, we use the usual *slice disorder measure* (SDM) [9] at time t, which is defined as

$$\text{SDM}(t) = \sum_{i \in A_t} |a_i(t) - e_i(t)| \, .$$

This metric is minimized at 0 when all estimates match the correct slice index.

In a distributed *slicing protocol* [9], nodes communicate via message-passing and estimate their own slice index during each time step. One of the metrics of interest is the message load incurred by a slicing protocol. A slicing protocol *converges* if it eventually provides a correct slicing of a static network, meaning that $\text{SDM}(t) = 0$ for all $t \geq t'$ for some t'.

As noted earlier, we are interested in protocols that are *(i)* simple, *(ii)* accurate, *(iii)* rapidly convergent, and *(iv)* efficient. With respect to efficiency, we will look at message load both in terms of the load experienced by participating nodes and the aggregated load on the network.

Let us illustrate the goal with a small example. Suppose the active nodes have attribute values 1, 2, 3, 7, 8, 9 and that $k = 3$. One way to slice the set is to sort the values (as shown), divide the list into three sets, and inform the nodes of the outcome. Alternatively, we might use an estimation scheme. For example, if node 7 obtains a random sample of the values, that sample might consist of 1, 7, and 9, enabling it to estimate the correct slice index. In the next section, we consider these and other options.

III. Sorting: One Step Toward Slicing

Intuitively, *sorting* and *slicing* are closely related problems. For this reason, it is not surprising that sorting algorithms have been used in prior slicing protocols [9], [17]. We discuss these approaches in this section.

A. Traditional Sorting Algorithms

Most parallel sorting algorithms [1], [3], [4], [7], [14], [21], [23] "wire together" nodes into a sort-exchange network, within which they can compare their value and decide whether to exchange their position. Such sorting networks are useful since they can provide nodes with indices in the sorted list, thus they make slicing possible. We explored a scheme for adapting parallel sorting algorithms to solve the slicing problem. This starts with a simple protocol to construct a spanning tree within which we were able to count the nodes in the system (n), number them consecutively, and route messages from node to node efficiently. The resulting overlay can then support any parallel sorting solution. However, this brings up the first drawback: in a P2P network, maintaining the overlay structure can be costly because of churn. Although our approach only needs $O(\log n)$ time to build the overlay, it needs to be rebuilt after any node joins or leaves.

On the other hand, in relatively static network parallel sorting solutions could be appealing. Ajtai, Komlós, and Szemerédi proposed the first algorithm to sort a system of n nodes in $O(\log n)$ steps. The big-O notation hides a large constant, which subsequent work has sought to decrease [6]; nonetheless, it remains over 1,000. Batcher's algorithm [4] has complexity $O(\log^2 n)$. Although an $O(\log n \log\log n)$-step algorithm is known [21], it competes with Batcher's solution only when $n > 2^{20}$. Other algorithms significantly reduce the convergence-time, sorting in $O(\log n)$ steps at the cost of achieving probabilistic precision. For example, an intuitive algorithm

known as the Butterfly Tournament [21] compares nodes in a pattern similar to the ranking of players during a tennis tournament. At the end of the algorithm each player has played $O(\log n)$ matches and can approximate his rank accurately.

We can thus expect a probabilistic parallel sorting algorithm to solve slicing exactly within time $O(\log n)$ in a static system. A deterministic sorting algorithm would need time $O(\log^2 n)$ to slice a static system. However, neither option seems to be viable in a P2P environment.

B. Sorting Attribute Values

We now discuss a technique used in a number of gossip-based slicing protocols [9], [17] which involves sorting nodes locally with respect to their attribute values. The key idea, as originally presented in [17], is the following. Each node initially chooses a random value (between 0 and 1) as an initial *position estimate*. Then, periodically, each node i searches its view $\mathcal{N}_i$ for misplaced neighbors, meaning that if i estimates its position to be ahead of its neighbor j, then i's attribute value should be less than j, and vice versa. If this is not the case, i swaps its position estimate with j. This process is repeated until all nodes are sorted.

The original Ordering algorithm [17] aims at reducing a global disorder measure, which expresses the distance of the current system state to a sorted state. The Ordering algorithm of [9] aimed at improving the original approach by reducing the slice disorder measure (defined in Section II-C). In the improved Ordering protocol, each node measures the disorder locally. This leads to a heuristic used by nodes to determine the best neighbor with which to swap position estimates, so as to maximize the reduction of disorder. Empirically, this variant improves on the original Ordering protocol [17].

More specifically, the Ordering protocol operates as follows. For a node i to evaluate the gain of exchanging with a node i' of its current view $\mathcal{N}_i$, we define the *local disorder measure* (LDM$_i$), which is equivalent to that defined in [9]. Let the *local attribute sequence* and the *local position estimate sequence* of node i be the ordered sequence of attribute values and position estimates, respectively, of all nodes in $\mathcal{N}_i$. These sequences are computed locally by i using the information $\mathcal{N}_i \cup \{i\}$. For any $i' \in \mathcal{N}_i \cup \{i\}$, let $\alpha_{i'}(t)$ and $\rho_{i'}(t)$ be the indices of $a_{i'}$ and $e_{i'}$ in the attribute sequence and the local position estimate sequence, respectively, of i at time t. At

any time t, the local disorder measure of node i is defined as

$$\mathrm{LDM}_i(t) = \sum_{i' \in \mathcal{N}_i(t) \cup \{i\}} (\alpha_{i'}(t) - \rho_{i'}(t))^2.$$

At time t, the heuristic considers each neighbor i' with which it could exchange its random value, and picks the one that maximizes the local gain $\mathrm{LDM}_i(t) - \mathrm{LDM}_i(t+1)$.

C. Limitations of Sorting

The sorting algorithms do not solve the slicing problem unless a sorted network gives the nodes their attribute value index. For slicing, each node needs to know its position relative to other nodes in the system. In many sorting algorithms, each node learns its predecessor and its successor, but not its absolute position in the sort order. As a consequence, sorting solutions do not necessarily solve the slicing problem.

Recall that the key idea in both Ordering protocols is to use a random number as a position estimate that is exchanged between nodes. Initially, every node chooses a random number as its position estimate, then each node compares periodically its estimate with that of a randomly selected peer. Since the initial random numbers will be used as the final position estimates of the nodes, if those numbers are not uniformly distributed, the final slice estimate is inaccurate. This can be a serious problem because in general, random position estimates may be far from uniform. The problem becomes even more severe if membership churn is correlated with the attribute values.

As an example in a three-node network, suppose that the initial random numbers of three nodes are 0.1, 0.15, and 0.2 but that a uniform distribution would have yielded values 0, 0.5, and 1. When the parallel sort terminates, all three will believe they belong to the first half of the system. In other words, slice estimates in the Ordering protocols may be incorrect even when the sorting phase terminates with the random values in a correct sort order. In the following section we describe solutions where this problem does not arise: protocols that guarantee an eventual optimal assignment in a static environment.

IV. Accurate Slicing

This section presents accurate solutions to the distributed slicing problem. We start by describing the Ranking protocol with eventual convergence guarantee, and then outline some of

its shortcomings. We then introduce Sliver: a new, simple algorithm that converges rapidly.

A. Ranking: Slicing Eventually

The so-called Ranking protocol was introduced in [9]. In contrast to the Ordering protocols of the previous section, a Ranking protocol does not assign unalterable random values as initial position estimates. Instead, nodes improve their position estimate each time new information is received. This reduces the slice disorder by a positive amount and eventually slices the system.

The Ranking protocol works roughly as follows. Periodically each node i updates its view $\mathcal{N}_i$ following an underlying protocol that provides a uniform random sample (e.g., [12], [18]). Node i computes its position estimate (and hence the estimate of its slice index) by comparing the attribute value of its neighbors to its own attribute value. The estimate is computed as the ratio of the number of lower attribute values that i has seen over the total number of attribute values i has seen.

Periodically i sends a message to some neighbors. There are two ways for node i to choose the destinations of its message. Either node i sends its message to a subset of neighbors from its current view $\mathcal{N}_i$ or i sends one message to each of the neighbors present in its view.

The first technique is used by the original Ranking protocol [9] and is as follows. Node i looks at the position estimate of all its neighbors. Then, i selects the node i' closest to a slice boundary (according to the position estimates of the neighbors of i). Node i also selects a random neighbor i'' among its view. Now, i sends an update message to i' and i'', containing its attribute value. The reason why a node close to the slice boundary is selected as one of the destinations is that such nodes need more samples to accurately determine which slice they belong to. This technique introduces a bias towards them, so they receive more messages.

The second technique is simple and speeds up the convergence at the price of additional messages: each node sends the message to all nodes present in its current view $\mathcal{N}_i$. Later, for a fair comparison of our protocol performance we adopt this technique to evaluate the Ranking protocol.

With either technique, upon reception of a message from node i, i' and i'' compute their new position estimate $\hat{p}_{i'}$ and $\hat{p}_{i''}$ depending on the attribute value received. The estimate of the slice a node belongs to follows the computation of the position estimate. Messages are transmitted using an asynchronous, one-way protocol, resulting in identical message complexity to the Ordering

protocols.

B. The Limitations of Ranking

The Ranking protocol will eventually converge if messages are received from nodes distributed uniformly among all nodes in the system. Observe that the protocols converge even if each node sends message preferably to some nodes (as in the original Ranking algorithm). Nevertheless, incoming messages must come from random neighbors distributed uniformly among all nodes. If the number k of slices is low then the system can be sliced rapidly, however, if k is large then the time taken to converge can be arbitrarily long. The underlying issue is that the precision of the estimates is tightly related to the degree of uniformity in the distribution of received attribute values.

Upon message reception, each node i estimates its position by comparing the attribute values of its neighbors with its own attribute value. It then estimates its position (and hence its slice index) as the ratio of the number of smaller attribute values that i has seen over the total number of values i has seen. As the algorithm runs, the position estimate improves. However, in the ranking protocol node i does not keep track of the nodes from which it has received values, thus, two identical values sent from the same node i' are treated by i as coming from two distinct nodes. In part for this reason, while the ranking protocol may converge eventually if the sending nodes are uniformly distributed among all nodes, it may not converge if the sending nodes embody any form of non-uniformity.

C. Sliver: Fast Slicing

We now introduce Sliver, a simple distributed slicing protocol which samples attribute values from the network and estimates the slice index from the sample. Sliver temporarily retains the attribute values and the node identifiers that it encounters. With this information, Sliver converges even if the neighbors distribution is skewed. Sliver reduces slice disorder rapidly: in Section V we show that slice estimates are expected to be close (off by at most one) with high probability after $O(\log n)$ time when $k = O(\log n)$, and $O\left(\sqrt{k \log n}\right)$ time when $k = O(n)$ and $k = \Omega(\log n)$.

To address churn, Sliver also retains the time at which it last interacted with each node, and gradually discards any values associated with nodes that have not been encountered again within a prespecified time window. The timeout ensures that the amount of saved data is bounded,

because the communication pattern we use has a bandwidth limit that effectively bounds the rate at which nodes are encountered. Moreover, this technique allows all nodes to cope with churn, regardless of potential changes to the distribution of attribute values in the presence of churn.

The code running on each node in this scheme at every time step is as follows.

- Each node i sends its attribute value to the nodes of its view $\mathcal{N}_i$. It then changes its view to a new random set of c nodes using peer sampling [18] or random walks [12].
- Each node i keeps track of the values it receives, along with the sender i' and the time they were received, and discards value records that have expired.
- Each node i sorts the m values it currently stores. Suppose B_i of them are lower than or equal to a_i.
- Each node i estimates its position as B_i/m and the slice index is estimated as the closest integer to kB_i/m.

Conceptually, Sliver is similar to the Ranking protocol. They differ in that nodes in Sliver track the node identifiers of the values they receive, whereas nodes in the Ranking protocol only track the values themselves. This change has a significant impact: Sliver retains the simplicity of the Ranking algorithm, but no longer requires that the sending nodes have a uniform sample of attribute values in the network as a whole.

We also consider a gossip-based variant of Sliver. Instead of having nodes only forward their *own* attribute values to other nodes, this variation also forwards current attribute values for *other* nodes (along with their sender identifier and the time since last confirmed update). To bound the use of memory and network bandwidth, we retain only the most recent R values in memory. For simplicity and also the sake of analysis, we consider only the original version of Sliver unless otherwise specified.

V. Theoretical Analysis of Sliver

Recall that Sliver stores recent attribute values and node identifiers it encounters in memory. At any point in time, each node can estimate its slice index using the current distribution of attribute values it has stored. We show analytically that if the system becomes synchronous then relatively short time has to pass for this estimate to be representative for all nodes. More precisely, we derive an analytic upper bound on the expected running time of the algorithm until

each node knows its correct slice index (within one) with high probability.

A. Assumptions

We focus on a static and synchronous system with n nodes and k slices, and we assume that there is no timeout, so that all values/identifiers encountered are recorded. The analysis can be extended to incorporate the timeouts we introduced to battle churn and to adapt to distribution changes, but may not offer as much intuition for the behavior and performance of the algorithm.

For the sake of simplicity, we assume that each node receives the values of one other randomly selected node in the system ($c = 1$) at each time step, and that all nodes start the protocol at time $t = 1$. Clearly, if a node collects all n attribute values it will know its exact slice index. A node i is *close* if it knows its slice index within at most one, and *stable* at time t if it remains close from time t henceforth. The problem lies with nodes whose whose position lies on the boundary of two slices. By considering stable nodes instead of exact estimates, we can derive meaningful results about the asymptotic time required by the system to reach a very low global slide disorder.

B. Convergence to a Sliced Network

In the following we show that Sliver slices the network rapidly. We assume that $k = O(n)$, since the slicing problem for $k > n$ is uninteresting. The following theorem gives the expected time it takes to achieve stability with high probability.

Theorem 5.1: We expect all nodes to be stable with high probability after

$$O\left(\sqrt{\max\{k, \log n\} \log n}\right)$$

time steps.

Proof: Let $\varepsilon > 0$. Fix some node i with value a_i. We assume that node i receives a previously unknown attribute value in each *time step*. The possibility of receiving redundant values is addressed later.

Let B_t denote the number of values that are known after t time steps which are below or equal to a_i. The fraction B_n/n is the true fraction of nodes with lower or equal attribute values. Knowing this fraction is equivalent to knowing the correct slice index of node i. There are on

average n/k nodes per slice, so node i is close as long as it reports a normalized slice index within n/k of B_n/n. We will estimate the probability that at time t, B_t/t is within t/k of B_n/n.

One can visualize the process, which we coin the *P-process*, as follows. There are B_n balls marked red and $n - B_n$ marked blue. In each time step t, a ball is randomly picked from the remaining ones and discarded. If it is red, $B_{t+1} \leftarrow B_t + 1$, otherwise $B_{t+1} \leftarrow B_t$. The probability

$$\mathbb{P}[\text{red ball at time } t] = \frac{B_n - B_t}{n - t}$$

depends on the current distribution of ball colors. Denote this probability by p_t. To simplify the analysis, we will consider the *Q-process* in which a red ball is picked with probability $q_t = B_n/n$ in each time step and blue otherwise. Notice that if $B_t/t \leq B_n/n$ then

$$p_t = \frac{B_n - B_t}{n - t} \geq \frac{B_n - tB_n/n}{n - t} = \frac{B_n}{n} = q_t,$$

and similarly if $B_t/t \geq B_n/n$ then $p_t \leq q_t$.

Consequently, the P-process tends to move towards B_n/n in each time step, whereas the Q-process ignores the proximity entirely. Analogously, imagine a car driving on the B_n/n road in the $(0, 1]$ world. Under the P-process the driver is more likely to turn towards the road when off-roading, whereas under the Q-process the driver always attempts to drive more or less parallel to the road. More rigorously, for a fixed constant a we can show by induction over t that the probability of next estimate B_t/t falling in the interval $\left[\frac{B_n}{n} - a, \frac{B_n}{n} + a\right]$ is greater for the P-process than the Q-process, for which it suffices to compare p_t with q_t near the endpoints of the interval. The details are straightforward and left for the reader. This observation implies that the bounds we will derive for the deviation from B_n/n at time t for Q-process act as an upper bound for the P-process.

We see that under the Q-process, $\mathbb{E}[B_t] = \sum_{i=1}^{t} q_t = tB_n/n$, since the steps are independent and identically distributed. We will use the following variant of the Chernoff-bound [2]. Let $X_1, \ldots X_N$ be independent identically distributed 0-1 random variables with $X = \sum_{i=1}^{N} X_i$ and

$\mu = \mathbb{E}[X]$.

$$\mathbb{P}[X \leq (1-\delta)\mu] \leq \exp\left(\frac{-\mu\delta^2}{2}\right) \qquad 0 < \delta \leq 1$$

$$\mathbb{P}[X \geq (1+\delta)\mu] \leq \begin{cases} \exp\left(\frac{-\mu\delta^2}{2+\delta}\right) & \delta \geq 1 \\ \exp\left(\frac{-\mu\delta^2}{3}\right) & 0 < \delta \leq 1 \end{cases}$$

For the Q-process we now derive

$$\begin{aligned} & \mathbb{P}\left[\frac{B_t}{t} \leq \frac{\mathbb{E}[B_t]}{t} - \frac{t}{k}\right] \\ = \;& \mathbb{P}\left[B_t \leq \left(1 - \frac{nt}{kB_n}\right)\mathbb{E}[B_t]\right] \\ \leq \;& \exp\left(-\frac{\mathbb{E}[B_t](nt)^2}{2(kB_n)^2}\right) \\ = \;& \exp\left(-\frac{t^3 n}{2k^2 B_n}\right) \\ \leq \;& \exp\left(-\frac{t^3}{3k^2}\right) =: s_t. \end{aligned}$$

since $B_n \leq n$.

Letting $\delta_t = nt/kB_n$ we can similarly derive for $\delta_t \leq 1$ that

$$\begin{aligned} \mathbb{P}[B_t \geq (1+\delta_t)\mathbb{E}[B_t]] &\leq \exp(-\mathbb{E}[B_t]\delta_t^2/3) \\ &\leq \exp\left(-\frac{t^3}{3k^2}\right) =: r_t \end{aligned}$$

and for $\delta_t \geq 1$ that

$$\begin{aligned} \mathbb{P}[B_t \geq (1+\delta_t)\mathbb{E}[B_t]] &\leq \exp\left(\frac{-\mathbb{E}[B_t]\delta_t^2}{2+\delta_t}\right) \\ &\leq \exp(-\mathbb{E}[B_t]\delta_t/3) \\ &\leq \exp\left(-\frac{t^2}{3k}\right) =: r'_t. \end{aligned}$$

Note that $s_t = r_t$, and $r_t \geq r'_t$ iff $t \leq k$.

All nodes in the network gather information about neighbors independently. Thus the probability that all nodes are close at time t, i.e. B_t/t is within t/k from B_n/n, is at least

$$\begin{aligned}(1-\mathbb{P}[B_t \leq (1-\delta_t)\mathbb{E}[B_t]])^n \cdot (1-\mathbb{P}[B_t \geq (1+\delta_t)\mathbb{E}[B_t]])^n &\geq (1-s_t)^n(1-\max\{r_t, r'_t\})^n \\ &\geq (1-r_t)^{2n}(1-r'_t)^n.\end{aligned}$$

The probability that all nodes remain close from time t to n is at least

$$\begin{aligned}&\prod_{T=t}^{n}(1-r_T)^{2n}(1-r'_T)^n \\ \geq\ &\prod_{T=t}^{n}(1-r_t)^{2n}(1-r'_t)^n \\ \geq\ &(1-r_t)^{2n^2}(1-r'_t)^{n^2} \\ \geq\ &(1-\max\{r_t, r'_t\})^{3n^2}\end{aligned}$$

Using that $r_t \leq \frac{1}{m}$ when $t \geq \sqrt[3]{3k^2 \ln m}$ and $r'_t \leq \frac{1}{m}$ when $t \geq \sqrt{3k \ln m}$, the previous bound is at least $(1-1/m)^{3n^2}$ when t is at least

$$\max\{\sqrt[3]{3k^2 \ln m}, \sqrt{3k \ln m}\}.$$

Let

$$m = 1 - \frac{3n^2}{\ln(1-\varepsilon)} \tag{1}$$

which is clearly $O(n^2)$ for a fixed value of ε. Now, for $t \geq \tau$

$$\begin{aligned}&(1-\max\{r_t, r'_t\})^{3n^2} \\ \geq\ &\left(1-\frac{1}{m}\right)^{3n^2} \\ =\ &\left(1-\frac{1}{m}\right)^{(m-1)(-\ln(1-\varepsilon))} \\ \geq\ &(1/e)^{-\ln(1-\varepsilon)} = 1-\varepsilon\end{aligned}$$

by using the fact that $\left(1-\frac{1}{x}\right)^{x-1} \geq 1/e$ for $x \geq 2$.

We now address the assumption that each node receives a distinct attribute value in each round.

The classic *coupon collector* problem asks how many coupons one should expect to collect before having all x different labels if each coupon has one of x distinct labels. The answer is roughly $x\log(x)$. For our purposes, the coupons correspond to attribute values (n distinct labels) and we wish to know how many rounds it will take to collect t distinct ones. Let T_j denote the number of rounds needed to have j distinct coupons if we start off with $j-1$. Then T_j is a geometric random variable with $\mathbb{E}[T_j] = n/(n-j+1)$.

The total time expected to collect t distinct coupons is thus

$$\begin{aligned}\sum_{j=1}^{t} \frac{n}{n-j+1} &\leq n(\ln n - \ln(n-t)) + \eta \\ &= n\ln\left(\frac{n}{n-t}\right) + \eta.\end{aligned}$$

Here η is at most the Euler-Mascheroni constant which is less than 0.6.

We now make use of the fact that $k = O(n)$. Since $\ln(m) = O(\log n)$ using the m from equation 1, there exists some constant $\alpha < 1$ such that

$$\tau = \max\left\{\sqrt[3]{3k^2\ln m}, \sqrt{3k\ln m}\right\} \leq \alpha n$$

for large n. Notice that $\sqrt[3]{3k^2\ln m} \geq \sqrt{3k\ln m}$ iff $k \geq 3\ln m$.

Since $1-x \leq \exp(-x)$ for $x \geq 0$, we derive for $0 < x < 1$ that

$$\frac{1}{x}\ln\frac{1}{1-x} \leq \frac{1}{1-x}.$$

It follows that

$$\begin{aligned}n\ln\frac{n}{n-\tau} &= \tau\left(\frac{n}{\tau}\ln\frac{1}{1-\frac{\tau}{n}}\right) \\ &\leq \frac{\tau}{1-\frac{\tau}{n}} \\ &\leq \frac{\tau}{1-\alpha}\end{aligned}$$

	Parallel Sorting	Ordering	Ranking	Sliver
Accurate	**yes**	no	**yes**	**yes**
Efficient	**yes**	**yes**	**yes**	**yes**
Robust to churn	no	no	**yes**	**yes**
Handles non-uniformity	**yes**	no	no	**yes**
Convergence time	$O(\log^2 n)$	$O(\log s)$	$O\left(\frac{p(1-p)}{d^2}\right)$	$O\left(\sqrt{\max\{k, \log n\} \log n}\right)$ (for stability)

TABLE I
COMPARISON OF SOLUTIONS TO THE SLICING PROBLEM. HERE s IS THE NUMBER OF SUCCESSFUL POSITION EXCHANGES OF THE ORDERING PROTOCOL, p IS THE ESTIMATED NORMALIZED INDEX OF A NODE, d IS THE MAXIMAL DISTANCE BETWEEN ANY NODE AND THE SLICE BOUNDARY AS DEFINED IN THE RANKING PROTOCOL AND k (THE NUMBER OF SLICES) IS $O(n)$.

Hence we expect all nodes to be stable (remain close) with high probability after

$$\begin{aligned} \frac{\tau}{1-\alpha} + \eta &= O\left(\max\left\{\sqrt[3]{k^2 \ln n}, \sqrt{k \ln n}\right\}\right) \\ &= O\left(\sqrt{\max\{k, \log n\} \log n}\right) \end{aligned}$$

time steps. ■

The assumptions we use in the theoretical framework are rather strong (no churn, fixed attribute values over time, infinite memory, all nodes start the protocol simultanously) so to highlight the performance of the algorithm in practice, we make use of real-life churn traces in the experiments.

VI. PERFORMANCE EVALUATION

This section evaluates Sliver's performance. First, it compares the performance of Sliver to the performance of related solutions. Second, Sliver and the Ranking protocols are compared using a real trace of storage space on a distributed testbed. Finally, we evaluate scalability by simulated Sliver on thousands of nodes, using a realistic trace that embodies substantial churn.

A. Theoretical Complexity

We discuss the advantages and drawbacks of the sorting algorithms presented in Section III-A, Ordering protocols presented in III-B, the Ranking protocols of Section IV, as well as our Sliver

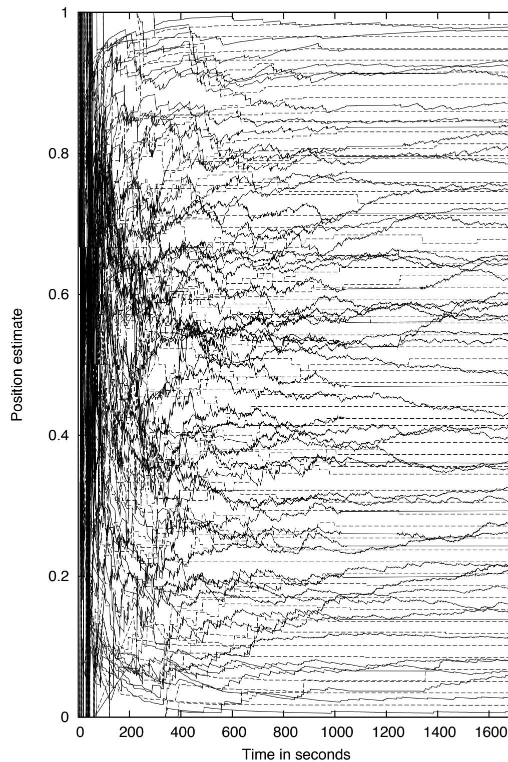


Fig. 1. Comparison of Sliver and the Ranking protocol for determining positions. Solid lines represent the positions given by the Ranking protocol over time, while dashed lines represent the positions given by the Sliver algorithm over time.

protocol.

These protocols differ in many ways, making it is difficult to give a precise comparison. Their complexity guarantees depend on a range of different parameters, such as the distance between the position of nodes and their closest slice boundary, and the number of successful position exchanges that occur during the execution. Despite these complexities, some observations are particularly interesting. We show details of the comparison in Table I.

Notice that, were it not for dynamicism, parallel sorting would have the best complexity: an impressive $O(\log^2 n)$ convergence guarantee; Sliver can only compete with this for small values of k, namely $k = O(\log^3 n)$ in a static system. However, most of the scenarios of interest to us are far more dynamic, and in such settings, as we will now see, Silver shines.

B. Distributed Experimentation

1) Sliver performance: We ran an experiment on 90 machines of Emulab [26], all of which run either RedHat Linux or FreeBSD. The Sliver protocol was executed among 60 machines while the 30 remaining machines were emulating the physical layer to make communication latencies realistic.

We implemented Sliver using GossiPeer [15], a framework that provides a low-level Java API for the design of gossip-based protocols. The underlying communication protocol is TCP and the average latency of communication has been chosen to match real latencies observed between machines distributed all over the world in PlanetLab. Additionally, we used storage information extracted from a real data set. The distribution of storage space used on all machines follows the distribution of 140 millions of files (representing 10.5 TB) on more than 4,000 machines [8].

This experiment seeks to slice the network according to the amount of storage space used. This sort of slicing would be of great interest in file sharing applications where the more files a node has, the more likely it is to be useful to others. To bootstrap the protocol, we provide each machine with the addresses of a few (five) others. While running, each node discovers new neighbors by executing random walks.

Figure 1 compares the performance of Sliver and the Ranking protocol [9] in the settings mentioned above with a timeout of 30 minutes and $c = 1$. The curves represent the evolution of the position estimate over time on each of these 60 machines for both protocols. (Four curves representing the nodes with the lowest position 0 and the largest position 1 are hidden at the bottom and top edges of the figure.) Note that each node can easily estimate the slice to which it belongs using this position estimate, since it knows the total number of slices k.

At the beginning of the experiment, all nodes have their position estimate set to 0, and time 0 represents the time the first message is received in the system. In the Sliver protocol, a majority of nodes know their exact position after 1,000 seconds and remain stable. In contrast, observe that with the Ranking protocol, even if no nodes join or leave, the random walks may not sample enough nodes to rapidly get a precise position estimate. As a result, even at 1700 seconds, no node knows its exact position with the Ranking protocol. Moreover, the estimates remain unstable. Since Sliver keeps track of the identity of the sending nodes, it stabilizes as soon as the values are known. Consequently in a larger system, even if the number of slices is linear in the system size (e.g. $k = n$), each node would know the slice to which it belongs.

This position is exploitable by a node to determine its slice. Depending on the portions of

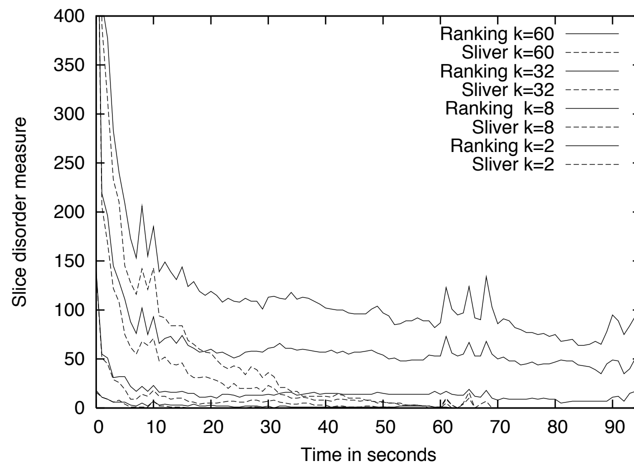
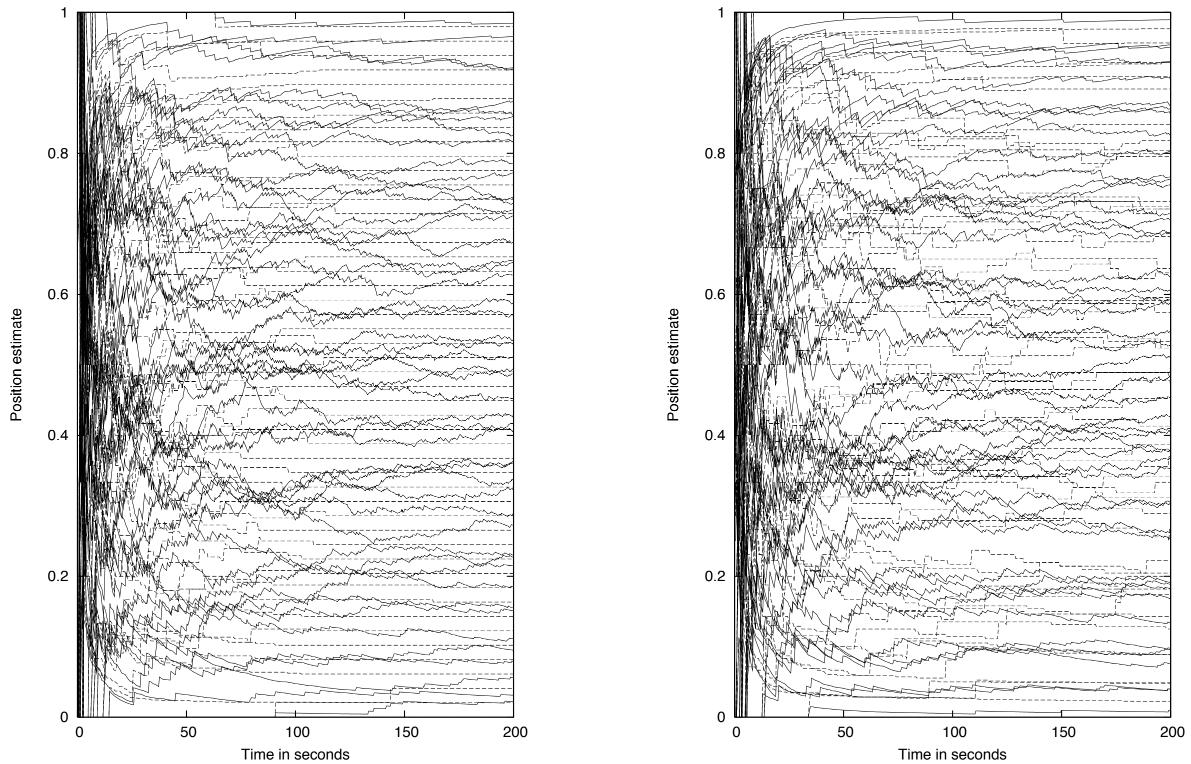


Fig. 2. Convergence time depending on the number of slices k.

the network the slices represent, the position can be approximated to a better or worse accuracy. For instance, if slices represent small portions (i.e., the number k of slices is large), then the position must be learned to high accuracy to compute the correct slice estimate. In contrast, if slices represent large portions (i.e., k is low), then a rough approximation of the position suffices to determine the right slice.

To better understand the impact of approximating the position on determining the slices, we illustrate how fast nodes using Sliver determine their slice index compared to the Ranking protocol while varying the number of slices k. Figure 2 indicates the slice disorder measure obtained by the Ranking and the Sliver protocols on the exact same experiment. Recall that the slice disorder measure is the sum over all nodes of the distance between the correct slice and the slice to which the node believes it belongs. The first observation is that in both protocols the convergence slows down as the number of slices enlarges. As mentioned previously, if k grows larger, then the portion represented by each slice shrinks; this forces us to compute a finer approximation of the positions, hence requires longer execution time. The second observation is that for varying number of slices k, the Sliver protocol reduces the slice disorder measure more rapidly than the Ranking protocol. For instance, after 60 seconds, the slice disorder measure obtained with the Ranking and the Sliver protocols are respectively 1 and 0 when $k = 2$, and are respectively 87 and 0 when $k = n$.



(a) Close or distant neighbors are chosen with the same probability. (b) Close neighbors are chosen preferably to distant neighbors.

Fig. 3. Comparison of Sliver and the Ranking protocol for determining positions depending on the choice of neighbors.

2) Geographical Network: Up to now, we have focused on experiments in which link delays were similar. To confirm that this did not bias our conclusions, we tested Sliver on a 50 node network composed of two distant datacenters communicating through long latency links hosting 25 machines each. In these experiments, the network has been configured using a NS2 configuration file as two LANs separated by WAN links. The communication latency intra-LAN is about couple of milliseconds whereas the communication latency inter-LAN is three hundred milliseconds. The view size has been extended to $c = 5$. As before, these experiments aim at slicing the system along the storage space metric extracted from the real trace of 140 millions of files.

We implemented a biased version of Sliver where each node communicates preferentially with nearby nodes and rarely with distant nodes. This reduces demand on the WAN links and is a common technique in gossip systems. As a result we obtain two different versions of the

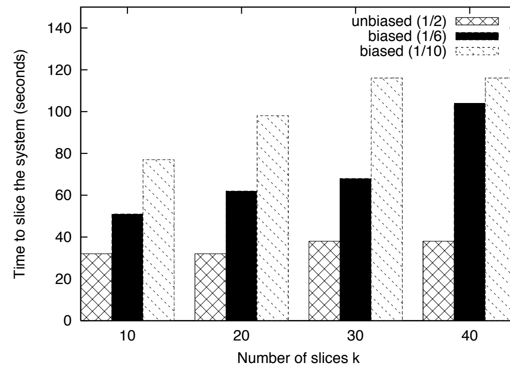


Fig. 4. Impact of the way neighbors are chosen on Sliver convergence time, as the number of slices k varies.

Sliver algorithm: the original unbiased one described earlier, and this new biased version. In the unbiased version, namely $p2$, when sending messages the probability that node i communicates with a node j of the distant LAN is identical to the probability of communicating with a local node: $\frac{1}{2}$. In the biased version, namely $p10$, the probability that a node i of communicates with a distant node is reduced to $\frac{1}{10}$.

The results for the unbiased version appear in Figure 3(a) and the results for the biased version appear in Figure 3(b). As expected and due to view size extension, we observe that the time taken to converge is shorter than in the former experiment. It is noteworthy that the time taken to converge is generally longer in the biased version than in the unbiased version. Even though the attribute values are uniformly distributed, communicating mostly with nearby nodes does not seem to shorten convergence time.

In order to get a closer look at the impact of neighbor choice on the convergence speed, we compared the convergence time of two distinct biased protocols and the unbiased protocol while slicing the network into $k = 10$, $k = 20$, $k = 30$, and $k = 40$ slices. In the first biased protocol, distant nodes are chosen with probability $\frac{1}{6}$ and close nodes are chosen with probability $\frac{5}{6}$, whereas in the second biased protocol, distant nodes are chosen with probability $\frac{1}{10}$ and close nodes are chosen with probability $\frac{9}{10}$. As before, the unbiased protocol is simply Sliver as originally presented (i.e., all nodes are chosen with the same probability). The results are shown in Figure 4. As expected, Sliver's convergence time is longer when the number k of slices is large. Preference for mearby nodes does not reduce convergence time, which confirms our former result. Nevertheless, biased versions multiply the convergence time of the unbiased

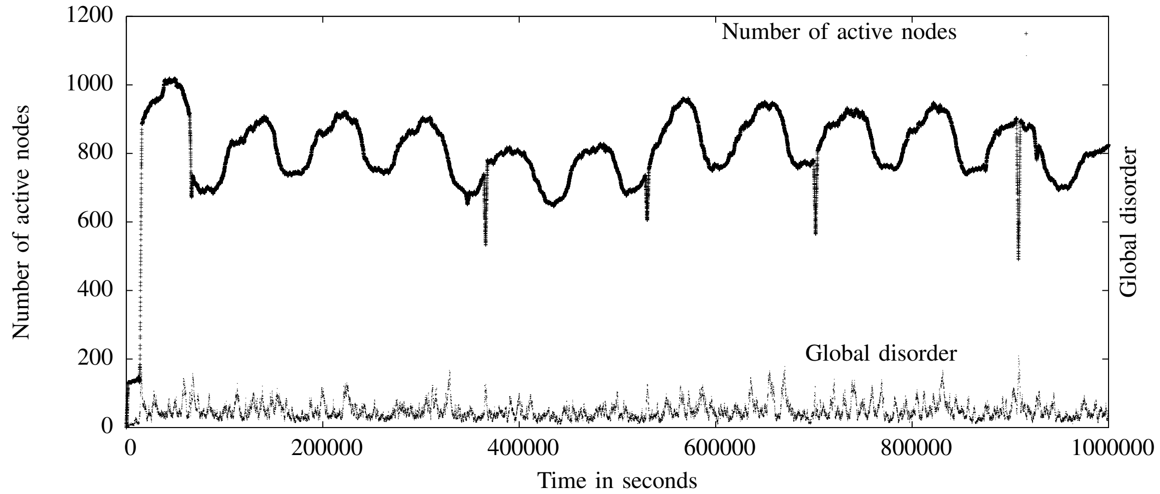


Fig. 5. Slice disorder measure of Sliver in a trace of 3,000 Skype peers.

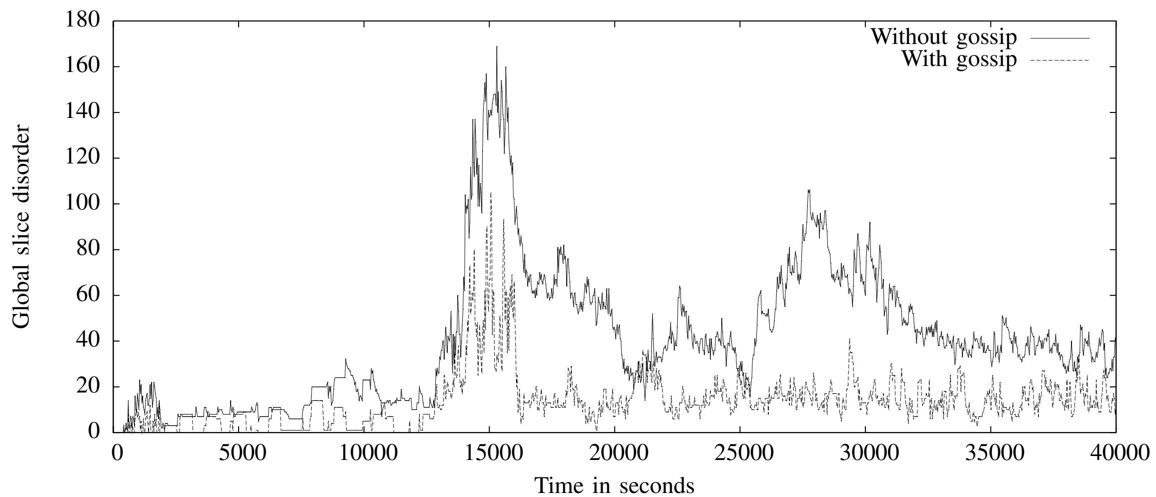


Fig. 6. Global slice disorder of Sliver with and without gossip support on the first 40,000 seconds of the Skype trace.

version by a factor of at most 3 for various values of k. In situations where a WAN link is very slow, one might accept this delay for greatly reduced WAN traffic.

C. Churn in the Skype Network

Next, we explored the ability of Sliver to tolerate dynamism in a larger scale environment. We simulated the Sliver protocol on a trace from the popular Skype VoIP network [24], using data that was assembled by Guha, Daswani, and Jain [16]. The trace tracks the availability of 3,000

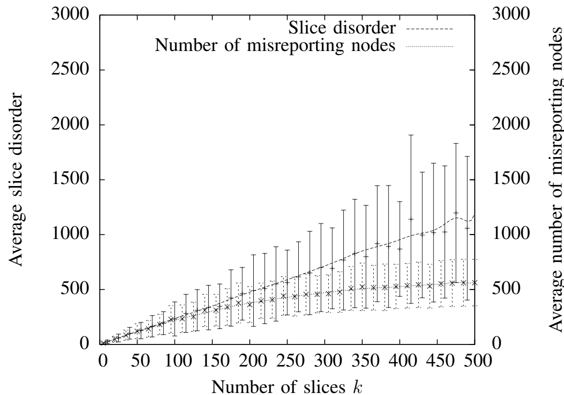


Fig. 7. Average slice disorder and number of misreporting nodes over the first 100,000 seconds in the Skype trace of 3,000 nodes as a function of the number of slices. Error bars represent one standard deviation.

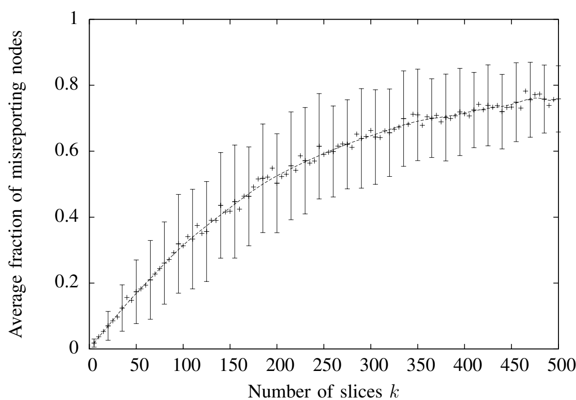


Fig. 8. Fraction of misreporting nodes at every step averaged over the first 100,000 seconds in the Skype trace as a function of the number of slices. Error bars represent one standard deviation.

nodes in Skype between September 1, 2005 to January 14, 2006. Each of these nodes is assigned a random attribute value and we evaluate our slicing protocol under the churn experienced by the nodes in the trace.

The goal of this experiment is to slice $n = 3,000$ nodes into $k = 20$ slices. We assume that every node sends its attribute value to $c = 20$ nodes chosen uniformly at random every 10 seconds. Attribute values that have not been refreshed within $5,000$ seconds are discarded. The top curve in Figure 5 shows the number of nodes that are available at a given point in time. The results show that on average less than 10% of the active nodes at any given time report a slice index off by one (or more), and the network quickly converges to have very low slice disorder.

Figures 7 and 8 illustrate the sensitivity of convergence time to k, the number of slices; the results are within the analytic upper bounds derived in Section V. For each of these figures, we ran the algorithm continuously within the Skype trace, identified erroneous slice estimates, and then averaged to obtain a “quality estimate” covering the full 100,000 seconds of the trace. Each node gossips every 10 seconds. Modifying this parameter effectively scales the convergence time by the same amount. We discard values that have not been refreshed within the last $5,000$ seconds. Note that when churn occurs, or a value changes, any algorithm will need time to react, hence the best possible outcome would inevitably show some errors associated with this lag.

We also evaluated the gossip-based modification of Sliver on the Skype trace, to evaluate its quality when faced with heavy churn. Each node retains at most $R = 300$ records of other values at any time, accounting for 10% of the total number of nodes. We used a timeout of $t = 500$ seconds, and nodes communicate every 10 seconds. In Figure 6 we compare the original Sliver protocol to this new variant by considering the slice disorder over time in the first 40,000 seconds. We can see that gossip version has 33% lower slice disorder on average, and rarely exceeds the disorder produced by the original Sliver algorithm. This experiment suggests that by using gossip, nodes can learn attribute values faster than without it, and the spread of records for nodes that have left the system (“ghost” values) that occurs in gossip does not have a significant impact on the quality of the estimate.

VII. Conclusion

This article evaluates a number of possible solutions to the distributed slicing problem. We introduce a new protocol called Sliver, and offer an analysis of its convergence properties. Some prior protocols converge slowly, some do not guarantee accuracy, and some are too easily disrupted by churn. We believe that Sliver offers the best overall balance of simplicity, accuracy, rapid convergence and robustness to churn.

REFERENCES

[1] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in c log n parallel steps. *Journal Combinatorica*, 3(1):1–19, March 1983.

[2] Dana Angluin and Leslie G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18(2):155–193, 1979.

[3] Friedhelm M. Meyer auf der Heide and Avi Wigderson. The complexity of parallel sorting. *SIAM J. Comput.*, 16(1):100–107, 1987.

[4] K.E. Batcher. Sorting networks and their applications. In *AFIPS 1968 Sprint Joint Comput. Conf.*, volume 32, pages 307–314, 1968.

[5] M. Blum, R. Floyd, V. Pratt, R. Rivest, and R. Tarjan. Time bounds for selection. *J. Computer and System Sciences*, 7:448–461, 1972.

[6] V. Chvátal. Lecture notes on the new AKS sorting network. Technical report, Rutgers University, 1992.

[7] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[8] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *Proc. of the 1999 ACM SIGMETRICS Int'l conference on Measurement and modeling of computer systems*, pages 59–70, 1999.

[9] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Raynal. Distributed slicing in dynamic systems. In *Proc. of the 27th Int'l Conference on Distributed Computing Systems*, 2007.

[10] Robert W. Floyd and Ronald L. Rivest. Expected time bounds for selection. *Commun. ACM*, 18(3):165–172, 1975.

[11] A.-T. Gai, F. Mathieu, F. de Montgolfier, and J. Reynier. Stratification in P2P networks: Application to BitTorrent. In *Proc. of the 27th Int'l Conference on Distributed Computing Systems*, page 30, 2007.

[12] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, Erwan Le Merrer, and Laurent Massoulié. Peer counting and sampling in overlay networks based on random walks. *Distributed Computing*, 20(4):267–278, 2007.

[13] Gnutella homepage. http://www.gnutella.com.

[14] Michael T. Goodrich. Communication-efficient parallel sorting. *SIAM J. Comput.*, 29(2):416–432, 1999.

[15] Vincent Gramoli, Erwan Le Merrer, and Anne-Marie Kermarrec. GossiPeer. `http://gossipeer.gforge.inria.fr`.

[16] S. Guha, N. Daswani, and R. Jain. An experimental study of the Skype peer-to-peer VoIP system. In *Proc. of the 5th Int'l Workshop on Peer-to-Peer Systems*, 2006.

[17] Márk Jelasity and Anne-Marie Kermarrec. Ordered slicing of very large-scale overlay networks. In *Proc. of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 117–124, 2006.

[18] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

[19] KaZaA homepage. http://www.kazaa.com.

[20] David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregrate information. In *Proc. of 44th Annual IEEE Symposium of Foundations of Computer Science*, pages 482–491, 2003.

[21] F. Thomson Leighton. *Introduction to parallel algorithms and architectures: array, trees, hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[22] Stefan Saroiu, Krishna P. Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, volume 4673, pages 156–170, 2002.

[23] Nir Shavit, Eli Upfal, and Asaph Zemach. A wait-free sorting algorithm. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 121–128, New York, NY, USA, 1997. ACM Press.

[24] Skype homepage. `http://www.skype.com`.

[25] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Internet Measurement Conference*, pages 189–202, 2006.

[26] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI '02: Proceedings of the 5th Symposium on Operating System Design and Implementation*, pages 255–270, 2002.

[27] N. E. Young. K-medians, facility location, and the Chernoff-Wald bound. In *Proc. of the 11th annual ACM-SIAM symposium on Discrete algorithms*, pages 86–95, 2000.