# Slicing Distributed Systems

Vincent Gramoli, Ymir Vigfusson, Ken Birman, Anne-Marie Kermarrec, and Robbert van Renesse

***Abstract*—Peer-to-peer (P2P) architectures are popular for tasks such as collaborative download, VoIP telephony, and backup. To maximize performance in the face of widely variable storage capacities and bandwidths, such systems typically need to shift work from poor nodes to richer ones. Similar requirements are seen in today's large data centers, where machines may have widely variable configurations, loads and performance. In this paper, we consider the *slicing* problem, which involves partitioning the participating nodes into $k$ subsets using a one-dimensional attribute, and updating the partition as the set of nodes and their associated attributes change. The mechanism thus facilitates the development of adaptive systems. We begin by motivating this problem statement and reviewing prior work. Existing algorithms are shown to have problems with convergence, manifesting as inaccurate slice assignments, and to adapt slowly as conditions change. Our protocol, *Sliver*, has provably rapid convergence, is robust under stress, and is simple to implement. We present both theoretical and experimental evaluations of the protocol.**

***Index Terms*—Distributed Systems, Fault Tolerance, Performance evaluation of algorithms and systems.**

## I. Introduction

Peer-to-peer (P2P) protocols are widely used for purposes such as building VoIP overlays and sharing files or storage. In principle, by exploiting the bandwidth and storage of participating nodes, P2P systems can achieve high performance without requiring an expensive data center. However, these systems struggle with a problem not seen in conventional client-server architectures: peers can be notoriously unpredictable and highly heterogeneous. For example, early versions of the Gnutella file-sharing system assigned roles to nodes without regard to their bandwidth and storage capacity. Experience revealed that even a few degraded nodes were enough to disrupt the entire platform [11], [20].

To avoid such problems, successful P2P platforms generally assume that resources such as storage space and bandwidth resources exhibit heavy-tailed distributions [21], [24]. They incorporate mechanisms that classify nodes, and then match workload to peer capacity, revising the mapping as conditions evolve over time. For example in the Kazaa [17] file sharing service, peers with longer lifetime and greater bandwidth play a more active role. Skype [23], a peer-to-peer telephony application, avoids routing calls through nodes that are sluggish or have low bandwidth. BitTorrent [6] uses an incentive mechanism under which nodes cooperate more closely with peers that match their own performance. Our goal is to offer a standardized solution that could be applied in a wide range of such settings.

A *slicing* algorithm is a mechanism for organizing a set of nodes into $k$ groups (the *slices*), such that each node rapidly learns the index of the slice to which it belongs. Slicing is done with respect to a one-dimensional attribute.[1] For the class of P2P applications in which peers, in a decentralized manner, adaptively work towards a global objective by some form of proportional resource sharing, slicing should be appealing.

For example, with $k = 4$ the slicing service would organize the nodes into quartiles. With $n$ nodes and $k = n$, slicing sorts them. No assumptions are made about the distribution of attribute values. Our target environments are highly dynamic: nodes come and go (churn), and attributes can change rapidly.

We are not the first to study slicing. In [15], the authors describe a communication-efficient parallel sorting algorithm and present node classification as a possible application, but the approach makes assumptions that many systems would not satisfy. For example, it requires uniform random value distributions and is not able to tolerate churn correlated to attribute values. An accurate slicing algorithm called the *Ranking protocol* was presented in subsequent work [10], but with very slow convergence. In larger deployments, membership churn can prevent the protocol from stabilizing. Moreover, as we will show in our experimental section, this sensitivity can be a real problem even when no churn occurs: our experiments reveal that even under relatively benign conditions, Ranking sometimes reports incorrect slice estimates that continue to drift as the run gets longer.

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

(i) Efficient and accurate computation of slice indices.
(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.

The main contribution of this article is a new randomized algorithm, *Sliver* (*Slicing Very Rapidly*). At its core is a sampling technique that uses a bounded memory to avoid value duplication. We compare Sliver with other possible solutions to the problem, including both prior slicing algorithms and parallel sorting, which we adapt to slicing. These alternatives each fail to achieve one or more of our goals. In contrast, Sliver is simple, achieves our goals, and is very fast.

Vincent Gramoli is with EPFL and University of Neuchâtel, Switzerland.
Ymir Vigfusson, Ken Birman, and Robbert van Renesse are with Cornell University, NY.
Anne-Marie Kermarrec is with INRIA Rennes Bretagne Atlantique, France.

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.

[1] Although beyond the scope of this paper, our protocol can be extended to slice multiple attributes. The protocol is such that by simply extending each message to carry distinct fields, for each of the attributes of interest, a single pattern of message exchanges can drive concurrent, side-by-side, executions of the algorithm.

## 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 connected. 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 latest attribute value $a_{i'}$ of $i'$, and optionally the value that $i'$ estimates to be its position (defined below). 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.

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 $j^{th}$ 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 $j^{th}$ 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.

### 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 *slice disorder measure* (SDM) [10] at time $t$ :

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

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

In a distributed *slicing protocol* [10], 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 *static network* is a network with no membership churn and fixed attribute values. A slicing protocol *converges* if it eventually provides a correct slicing of a network that is static from some point on, 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. Approaches to Slicing

This section reviews prior protocols related to Sliver. These fall into two classes. The first group are solutions that use a sorting mechanism to drive a probabilistic slicing scheme, while the second class use sampling to estimate the attribute distribution (these depend upon the ability to do uniform sampling).

### A. Slicing by Sorting

Intuitively, *sorting* and *slicing* are closely related problems. In particular, if we are given a decentralized mechanism for sorting attribute values, so that each node can learn the number of nodes in the system and its index in the sort order, the computation of its slice becomes trivial. Below, we show that parallel sorting algorithms can be adapted to a P2P setting, and hence used to solve slicing. However, we will also see that the solutions are complex and potentially fragile in the face of churn.

A closely related option is to use randomized gossip-based sorting algorithms to drive a slicing mechanism. This has been explored in prior work, and yields approximate slice estimates [10], [15]. As we will see, however, these algorithms may fail to converge.

*1) Parallel Sorting on a P2P Overlay:* We begin by looking at the feasibility of using parallel sorting in P2P settings. Most parallel sorting algorithms [1], [3], [4], [7], [12], [18], [22] "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. 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 [5]; 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 [18], 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 [18] 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 estimate its rank accurately. In a static network, a probabilistic parallel sorting algorithm can solve slicing with high probability within $O(\log n)$ time. In contrast, a deterministic sorting algorithm would need time $O(\log^2 n)$ to slice a static system.

To convert a parallel sorting algorithm into a P2P slicing solution, one starts by constructing an overlay tree on the active nodes. It is not difficult to build a spanning tree, within which nodes can be counted and assigned identifiers in the range $1, \ldots, n$. Having done this, one can route messages from node $i$ to node $j$ along the tree. Of course, such a tree can be disrupted by churn, and this makes the approach more complex: to tolerate failures, both the tree itself and the attribute values must be replicated.

Having taken these steps, one can then run a parallel sorting algorithm on the overlay. If we sort tuples consisting of *(value,origin node)* pairs, and then report the final location of each value back to the origin node, each node learns its position in the sort order and hence its exact slice number. Although brevity precludes inclusion of a detailed analysis, tree construction brings an additional $O(\log n)$ cost, beyond the cost of the sorting algorithm itself. Thus the approach is feasible. However, the construction is complex, particularly because of the need to make the overlay robust to churn.

*2) Gossip-Based Parallel Sorting:* As noted earlier, one can also use a gossip-based sorting mechanism to drive a slicing protocol [10], [15]. In the remainder of this paper, we refer to these protocols as *Ordering* protocols, as suggested in [10]. The idea is for each node to choose a random value (between 0 and 1) as an initial *position estimate*. Then, in a series of randomized gossip exchanges, each node $i$ searches its view $\mathcal{N}_i$ for misplaced neighbors. In particular, 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.

Here, we focus on the Ordering protocol in [10], which improves on the one in [15]. The protocol works by having each node measure the slice disorder using local information. This leads to a heuristic used by nodes to determine the best neighbor with which to swap position estimates. 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

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

At time $t$, the Ordering protocol considers each neighbor $i'$ with which it could exchange its random value, and picks the one that maximizes the local gain $\mathrm{LDM}_i(t) - \mathrm{LDM}_i(t+1)$. However, while it is straightforward for each node to compute the LDM, doing so gives only a rough idea of the value of the global disorder measure. Accordingly, we make no further use of the LDM in what follows.

*3) Limitations:* 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. Recall that in our attempt to adapt parallel sorting algorithms to solve the slicing problem, we addressed this by sorting tuples and ultimately informing each node of the index at which its attribute ended up, in the sorted order. The Ordering protocol, however, lacks this kind of fine-grained information, and merely estimates the position, and this turns out to be a significant source of inaccuracy.

Recall that the key idea in the Ordering protocol 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 may be incorrect even when the sorting phase terminates with the random values in a correct sort order. Even if the initial distribution of random values is initially perfectly uniform, a variation in the distribution of attribute values leads also to incorrect slice estimates. In the following section we describe solutions where this problem does not arise.

### B. Slicing by Ranking

Next, we present Ranking, a recently proposed slicing protocol. Although the protocol is simple, we will see that it depends strongly upon the assumption that there is a good way to obtain uniform random samples of node values that improve steadily over time. In many settings this is not possible, and with non-uniform samples, the algorithm does not converge even in a static system.

*1) Ranking, Slicing Eventually:* The Ranking protocol was introduced in [10]. Unlike the Ordering protocols of the previous section, a Ranking protocol does not assign immutable 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., [16]). 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 [10] 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 speeds up the convergence at the price of additional messages: each node sends the message to all nodes present in its current view $\mathcal{N}_i$. We will use this technique when we 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.

*2) Limitations:* The Ranking protocol is not guaranteed to converge, even on a static network. 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. The event that $i$'s slice estimate is skewed because of receiving too many identical values below (or above) $a_i$ happens with positive probability and thus no convergence guarantee can be established. This crucial shortcoming will also be evident in our experimental work.

## IV. Sliver, Fast and Accurate Slicing

We now introduce Sliver, a simple distributed slicing protocol that samples attribute values from the network and estimates the slice index from the sample.

### A. Tracking Value Owners

Sliver temporarily retains the attribute values and the node identifiers that it encounters. With this information, Sliver is guaranteed to converge in a static network. Sliver reduces slice disorder rapidly: in Subsection IV-B 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 [16] or random walks [19].
- 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 protocol, but no longer requires that the sending nodes have a uniform sample of attribute values in the network as a whole. If no time-out is specified, we assume that it is infinite.

We also consider an indirect information tracking (IIT) 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.

We leave several topics for future study. These include support for unevenly balanced slice sizes, and protocols that fix the slice size, allowing $k$ to vary. A very interesting question concerns multi-dimensional attribute sets. As noted earlier, Sliver has an obvious generalization in which messages carry an array of information, one entry for each attribute, permitting the algorithm to concurrently slice in several dimensions. But one can also imagine schemes for combining sets of attributes to create lower-dimensional pseudo-attributes. Slicing might then function as a crude (but inexpensive) parallel clustering algorithm.

### B. Theoretical Analysis of Sliver

In this section we analyze the convergence properties 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. 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.

*1) 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 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 slice disorder.

*2) 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 4.1:* 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 $\varepsilon$. 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 $\eta$ 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. ■

From the analysis, we can see that the expected amount of memory on a node will be bounded by the number of rounds required for the protocol to converge. However, the theoretical analysis overlooks practical issues such as churn, evolution of attribute values over time, and nodes that aren't perfectly synchronized. We therefore undertook a series of realistic experiments that validate Sliver using real churn traces and attribute distributions, in Section IV-C.

*3) Discussion:* We are now in a position to compare the sorting-based protocols discussed in III-A, the Ranking protocol of Section III-B, as well as our Sliver protocol.

These protocols differ in many ways, making it 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 Sliver compares very favorably with parallel sorting. If there are very few slices ($k$ small relative to $n$), the initial slice estimate is likely to be correct for most nodes. Parallel sorting would be an absurdly complicated overkill. With larger values of $k$, the situation is less clear: When $k = \Omega(\log^3 n)$, the theory favors parallel sorting, but in our experiments Sliver converged rapidly even with $k = n/2$.

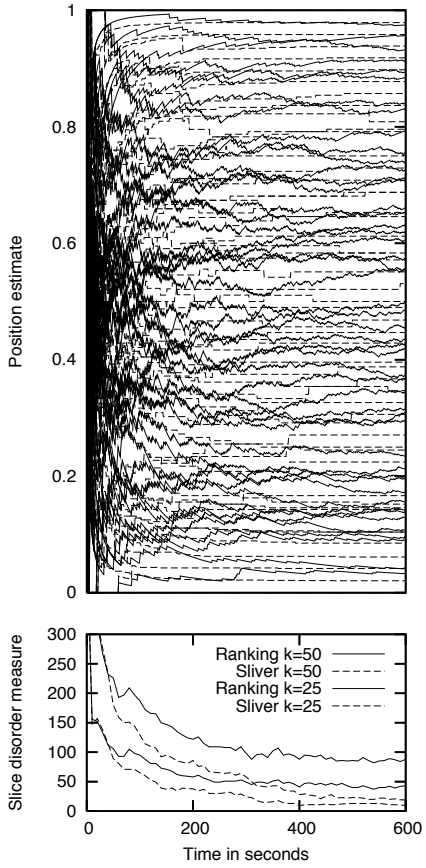


Fig. 1. Comparison of Sliver and the Ranking protocol for determining positions and slicing the network. Solid lines represent the position estimates and the slice disorder measure obtained with the Ranking protocol as a function of time (measured by tracking the number of messages received by each node), while dashed lines represent the positions and measure given by the Sliver protocol.

### C. Performance Evaluation

This section evaluates Sliver's performance. First, it compares the Sliver and Ranking protocols. We look at a LAN and then a WAN scenario, slicing with respect to a storage-space attribute. Next, we evaluate scalability by simulating Sliver on thousands of nodes, using a realistic trace that embodies substantial churn.

*1) Distributed Evaluation:* We performed experiments using 50 Emulab nodes that run Sliver and Ranking. Emulab [25] is a distributed testbed that allows the user to specify a specific network topology using NS2 configuration file. Using this feature, we set the communication delays to model realistic network latencies as observed in PlanetLab.

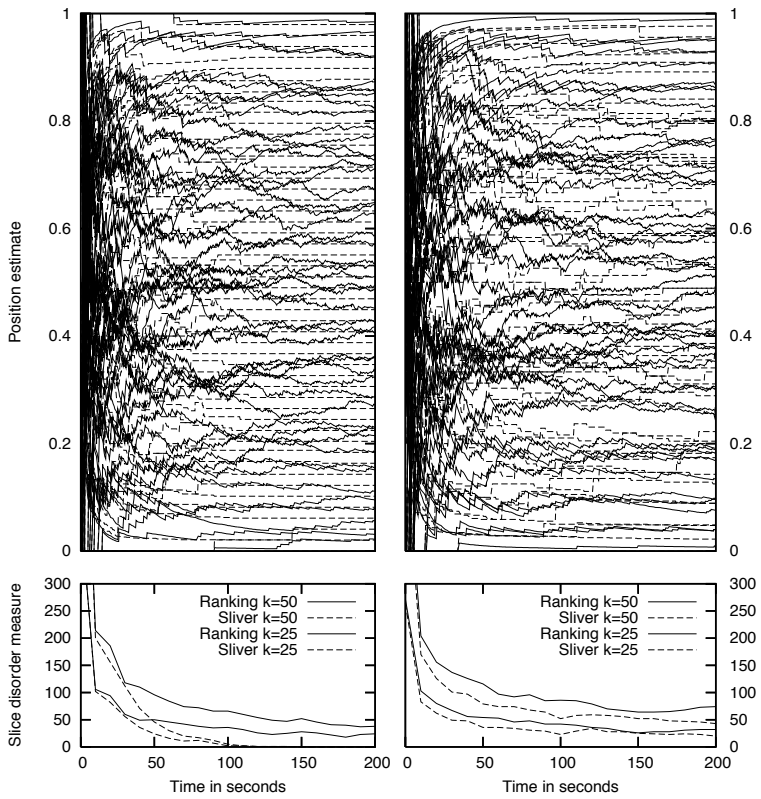


Fig. 2. Comparison of Sliver and the Ranking protocol for determining positions and slicing the network depending on the choice of neighbors. Left-hand side: close or remote neighbors are chosen with the same probability. Right-hand side: close neighbors are chosen preferentially.

Our experiment slices the network according to the amount of storage space used, a metric of obvious practical value. We implemented Sliver (not the IIT variant) side by side with the Ranking protocol, within the GossiPeer [13] framework, an emulation platform that provides a low-level Java API for experimentation with gossip-based protocols. For realism, the distribution of storage space used matches a distribution in a trace of 140 million files (representing 10.5 TB) on more than 4,000 machines [9]. Our 50 machines randomly sampled this distribution to set their storage attribute values.

*a) Ranking versus Sliver:* In the first experiment presented in Figure 1 all 50 nodes execute Sliver and the Ranking protocol in parallel (the pattern of peering is thus identical for Sliver and for Ranking; only the algorithm itself differs). To bootstrap, nodes are initialized with the addresses of 5 other nodes. Every node chooses $c = 1$ contact node every 2 seconds by executing a random walk of depth 4. Once the random target is reached, it responds directly to the initiator. The top graph of Figure 1 represents the position estimate of each of the 50 nodes as a function of time. Note that each node can easily evaluate the slice to which it belongs using this position estimate since each node knows the number of slices $k$. The bottom graph of Figure 1 shows the slice disorder measure evolution for $k = 25$ and $k = 50$ slices resulting from the above position estimates. It is evident that Sliver is more rapidly convergent than Ranking. For example, at time $400$ the Sliver position estimate is 3 times less disordered as that of the Ranking protocol.

We find it striking that Sliver is so much more stable than Ranking. This highlights the extreme sensitivity of Ranking to the node sampling technique. To the extent that peers are revisited by the random walk mechanism, the Ranking estimate may incorporate the same attribute values more than once, causing drift. Moreover, there is no particular reason that additional samples should improve the estimate. We see this as a significant issue for the Ranking protocol. Of course Sliver also suffers if a node experiences a poor quality of sampling, as evidenced by a few nodes that adjust their position estimates as late as time $500 - 600$. However this is really a different issue: here, the effect is due to non-uniform sampling of the underlying data distribution, not re-sampling of previously visited nodes.

*b) Geographical network:* The experiments in part (a) assumed constant link latencies, as might occur in a large data center. However, many P2P systems operate in WAN settings where this assumption would not be valid. In such settings, P2P systems very often bias gossip, selectively favoring nearby nodes (so as to benefit from reduced latency communication) and reducing the rate of gossip with remote nodes. Readers interested in the broader issues that arise here are referred to [8], where Demers and his colleagues offer a rigorous analysis of this form of biased gossip.

To evaluate the impact of bias on Sliver and Ranking, we set up an experiment to emulate a pair of data centers containing 25 nodes each, each node randomly picks storage attributes based on the distribution in our trace. Inter-node

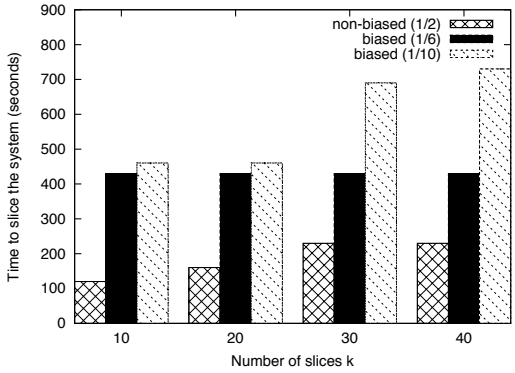


Fig. 3. Impact of the way neighbors are chosen on Sliver convergence time, as the number of slices $k$ varies.

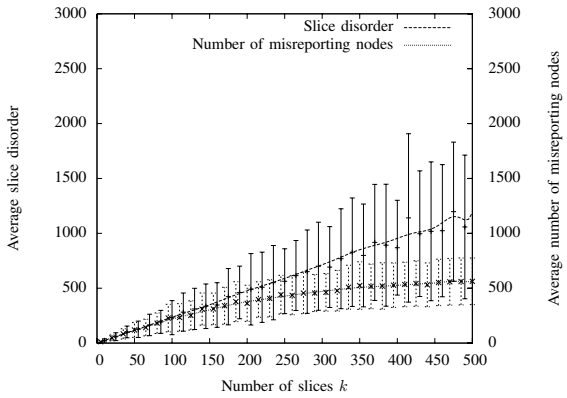


Fig. 6. Average slice disorder and number of misreporting nodes over the first 28 hours in the Skype trace of 3,000 nodes as a function of the number of slices. Error bars represent one standard deviation.

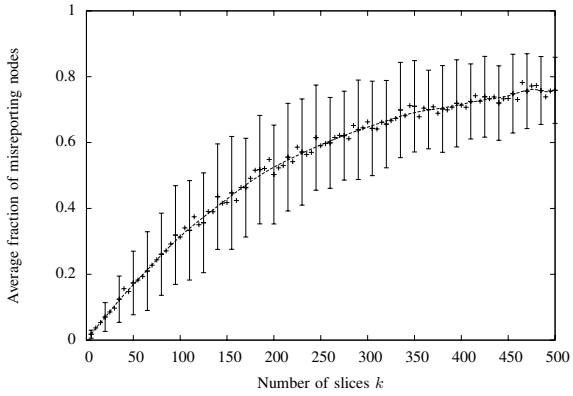


Fig. 7. Fraction of misreporting nodes at every step averaged over the first 28 hours in the Skype trace as a function of the number of slices. Error bars represent one standard deviation.

latencies were set to 2ms within the data centers, and to 300ms for communication over the WAN link between the centers. Each node gossips every second with $c = 1$ other node. We implemented a biased version of Sliver and Ranking where each node communicates preferentially with nearby nodes and rarely with remote nodes. In the unbiased version when sending messages the probability that node $i$ communicates with a remote node is $\frac{1}{2}$. In the biased version this probability drops to $\frac{1}{10}$. As expected, the unbiased experimental runs give essentially identical results to those seen above, so the interest here is in the comparison of unbiased to biased runs.

The results for the unbiased version appear in Figure 2 (left-hand side) and the results for the biased version appear in Figure 2 (right-hand side). Again, Sliver significantly outperforms Ranking. Notice also that for both algorithms, bias slows 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 Sliver protocols and the unbiased Sliver protocol while slicing the network into $k = 10$, $k = 20$, $k = 30$, and $k = 40$ slices. In the first biased protocol, remote nodes are chosen with probability $\frac{1}{6}$ and close nodes are chosen with probability $\frac{5}{6}$, whereas in the second biased protocol, remote 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 3. As expected, Sliver's convergence time is longer when the number $k$ of slices is large. Preference for nearby nodes does not reduce convergence time, which confirms our former result. However, the effect isn't extreme: the convergence time increases by a factor of at most 3 for the various values of $k$. We suspect that there are many settings in which this degree of slowdown could be tolerated to slash the load on an overburdened WAN link.

We should perhaps stress that all of these experiments used essentially the same distribution of attribute values for nodes in each of the two data centers. Had this not been the case (for example, if nodes in Bangalore happen to have very different amounts of storage than nodes in Ithaca), the effect of bias could be dramatic: both algorithms would begin to exhibit two distinct regions of convergence, first discovering local conditions, and then adapting as remote data trickles in.

*2) 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 [23], using data that was assembled by Guha, Daswani, and Jain [14]. The trace tracks the availability of 3,000 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 90 minutes are discarded. The top curve in Figure 4 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. Under such conditions, use of the IIT variant of Sliver would be highly advisable.

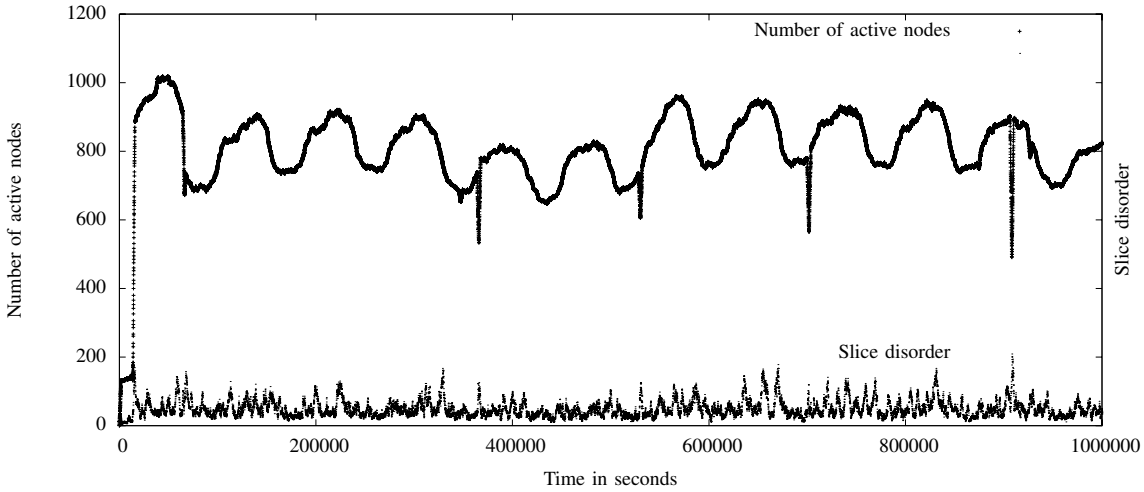


Fig. 4. Slice disorder measure of Sliver in a trace of 3,000 Skype peers.

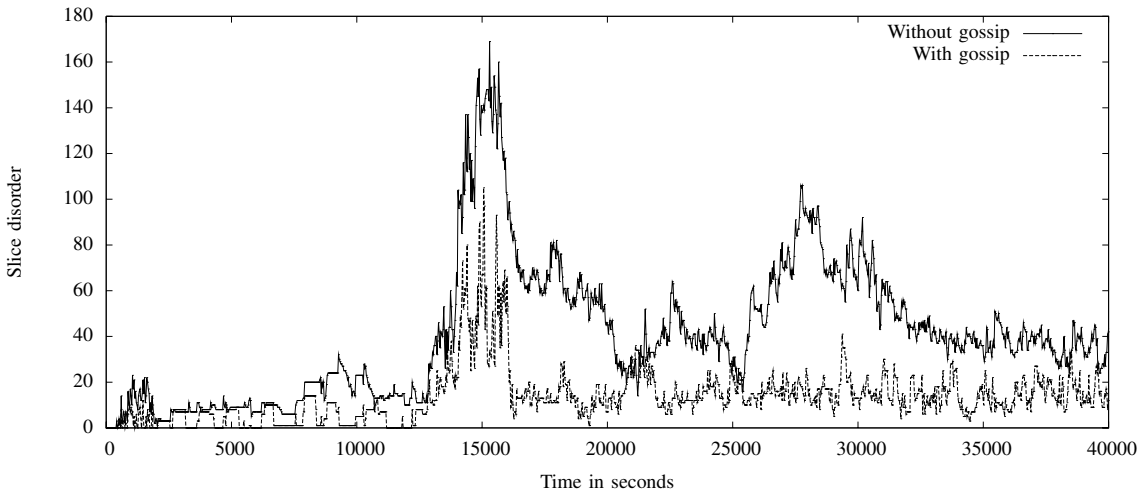


Fig. 5. Slice disorder of Sliver with and without indirect information tracking on the first 12 hours of the Skype trace.

Figures 6 and 7 illustrate the sensitivity of convergence time to $k$, the number of slices; the results are within the analytic upper bounds derived in Subsection IV-B. 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 28 hours 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 90 minutes. 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 IIT variant 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 5 we compare the original Sliver protocol to this new variant by considering the slice disorder over time in the first 12 hours. We can see that indirect information tracking reduces slice disorder by 33% on average, and rarely exceeds the disorder produced by the original Sliver protocol. This experiment suggests that by using indirectly acquired information, perturbed nodes learn attribute values more quickly. Moreover, the impact of records for nodes that have left the system ("ghost" values) is seen to be insignificant, at least in this experiment. Clearly, had churn somehow been correlated with attribute values (for example, if nodes low on storage space often crash), our findings might have been very different.

## V. Conclusion

The *slicing* problem organizes the nodes in a system into an integral number of groups, such that each node learns its *slice number*. For example, with four slices ($k = 4$), nodes are organized into quartiles. Slicing has many uses: the technique can help systems to route around overloaded nodes,

to assign special tasks to lightly loaded nodes, or to trigger adaptation as needed. Here, we started by evaluating existing slicing methods, and identified a number of limitations. Our analysis of prior work, and of an approach in which slicing is performed as a side-effect of running a parallel sorting algorithm, reveals that all of these options are inadequate in significant ways. Our Sliver protocol is remarkably cheap, achieves provably rapid convergence, is robust under various forms of stress, and is simple to implement.

## ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers, who helped improve the structure and contents of this paper.

## 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] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for hamiltonian circuits and matchings. *Journal of Computer and System Sciences*, 18(2):155–193, 1979.

[3] F. M. Meyer auf der Heide and A. Wigderson. The complexity of parallel sorting. *SIAM Journal on Computing*, 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] V. Chvátal. Lecture notes on the new AKS sorting network. Technical report, Rutgers University, 1992.

[6] B. Cohen. Abstract incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, 2003.

[7] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[8] A. J. Demers, D. H. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. E. Sturgis, D. C. Swinehart, and D. B. Terry. Epidemic algorithms for replicated database maintenance. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.

[9] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 59–70, 1999.

[10] A. Fernández, V. Gramoli, E. Jiménez, A.-M. Kermarrec, and M. Raynal. Distributed slicing in dynamic systems. In *ICDCS '07: Proc. of the 27th IEEE International Conference on Distributed Computing Systems*, 2007.

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

[12] M. T. Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

[13] V. Gramoli, E. Le Merrer, and A.-M. Kermarrec. GossiPeer. `http://gossipeer.gforge.inria.fr`.

[14] S. Guha, N. Daswani, and R. Jain. An experimental study of the Skype peer-to-peer VoIP system. In *Proceedings of the 5th International Workshop on Peer-to-Peer Systems*, 2006.

[15] M. Jelasity and A.-M. Kermarrec. Ordered slicing of very large-scale overlay networks. In *P2P '06: Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, pages 117–124, 2006.

[16] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3):8, 2007.

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

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

[19] L. Lovász. Random walks on graphs: A survey. *Combinatorics, Paul Erdös is Eighty*, 2:353–398, 1996.

[20] M. Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing*, page 99, Washington, DC, USA, 2001. IEEE Computer Society.

[21] S. Saroiu, K. P. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking*, volume 4673, pages 156–170, 2002.

[22] N. Shavit, E. Upfal, and A. 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.

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

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

[25] 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.