Sliver, a Fast Distributed Slicing Algorithm [1]

Vincent Gramoli*†, Ymir Vigfusson†, Ken Birman†, Anne-Marie Kermarrec‡, Robbert van Renesse† *EPFL - University of Neuchatel, Switzerland Email: vincent.gramoli@epfl.ch
†Cornell University, Ithaca, NY. Email: {ymir, ken, rvr}@cs.cornell.edu
‡INRIA Rennes Bretagne Atlantique, France. Email: akermarr@irisa.fr

Abstract

Slicing a distributed system involves partitioning the nodes into k equal-size subsets using a one-dimensional attribute. A new gossip-based slicing algorithm that we call *Sliver* is proposed here; relative to alternatives, it converges more rapidly to an accurate solution and does a better job of tolerating churn. The low cost and simplicity of the solution makes it appropriate for use in a wide range of practical settings.

Index Terms

Distributed slicing, Churn, Supernodes, Peer-to-peer.

I. Introduction

Peer-to-peer (P2P) protocols have emerged as the technology of choice for such purposes as building VoIP overlays and sharing files or storage. The algorithms discussed here are intended to assist such applications in dealing with heterogeneous, often heavy-tailed resource distributions [1], [2]. For example, VoIP applications such as Skype [3] must avoid routing calls through nodes that are sluggish or have low bandwidth, and file sharing services such as Gnutella [4] and Kazaa [5] employ a two-level structure, in which peers with longer lifetime and greater bandwidth capabilities function as *superpeers*.

A *distributed slicing* protocol organizes n nodes into k equally balanced *slices* in a one-dimensional attribute space, such that when the classification algorithm terminates, each node knows the slice number to which it belongs. Potential applications include:

- Construction of multi-level hierarchies (generalizing the two-level ones common today).
- Identifying outliers with respect to attributes indicative of possible malfunction or malicious behavior.
- Load-balancing in datacenters, or to provide varying levels of service in accordance with a classification of clients.

We are not the first to study slicing. In [6], the authors offer a communication-efficient parallel sorting algorithm and point to node classification as a possible application, but do not actually propose a slicing algorithm. Subsequent work [7] presented an accurate slicing algorithm (the *Ranking* protocol), but it converges very slowly: in larger deployments, churn would often disrupt the underlying system long before the algorithm stabilizes, and it might never catch up. Ideally, a slicing algorithm:

(i) Should efficiently compute accurate slice numbers, in a sense formalized below.
(ii) Should achieve provably rapid convergence.
(iii) Should be robust to membership churn and potentially rapid evolution of underlying attribute values.

For practical purposes, it is also preferable that the protocol be as simple as possible.

This work was supported, in part, by NSF, AFRL, AFOSR, INRIA/IRISA and Intel Corporation.
Manuscript created 12/10/2007. Revised 04/24/2008.

In what follows, we develop a new gossip-based algorithm we call *Sliver* (for *Slicing Very Rapidly*).[2] A rigorous analysis of convergence properties establishes the first known upper bound on expected convergence time as a function of k, the number of slices. We implemented Sliver, and report our findings for a trace-driven evaluation, using real data from the Skype VoIP overlay.

As a part of our effort, we reviewed other ways of solving the slicing problem. In the section on related work, we suggest a simple way of adapting parallel sorting algorithms for use in P2P settings. This gives us access to some very sophisticated options. However, such algorithms also bring substantial complexity, and churn would limit the size of problems that can be tackled using them. Neither problem arises with Sliver, which is relatively simple and tolerant of churn.

II. Model and Problem Definition

A. Problem Statement

The system consists of n nodes with unique 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. Nodes can join and leave (or fail by halting), hence the set of nodes and the value of n changes over time; we say that the nodes currently in the system are *active*. Each node i has an *attribute value* $x_i \in \mathbb{R}$ that represents its capacity in the metric of interest, for example uplink bandwidth. At any time, the *relative position* r_i of node i is the index of x_i within the sorted attribute values, normalized to fall within the range $[0,\ldots,1]$ (computed by dividing the index by n). If two nodes have identical values, we break ties using the node identifiers. The sets $P_1,\ldots,P_k$ partition $(0,1]$ and are called *slices*. Here, we focus on equally sized slices, that is, we set $P_j = (\frac{j-1}{k}, \frac{j}{k}]$ for any j, $0 < j \leq k$. Node i belongs to slice P_j if $r_i \in P_j$. Given a value of k, a slicing protocol [7] computes k slices such that upon termination, each node knows its slice membership. Nodes communicate by message-passing, hence message load will be a metric of interest. A slicing protocol *converges* if it eventually provides a correct slicing of a static network.

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.

A quick example will illustrate our goal. 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 number. In the next section, we consider these and other options.

III. Gossip Protocols

In a *gossip* protocol, nodes exchange bounded amount of information ("gossip") with a constant number of peers picked uniformly at random in every time step. Here, we discuss two prior gossip-based solutions to the slicing problem, identifying serious limitations to each. The analysis motivates the Sliver protocol.

The *Ordering* algorithm by Jelasity and Kermarrec [6] uses a randomized parallel sort implemented over a gossip-based communication substrate. The key idea is to have each node start by picking a random number as its initial estimate of its rank in the sort order, and then to adjust the rankings via gossip exchange: if node A gossips with node B and A has a smaller attribute value than B, A takes the smaller of the two ranking estimates and B the larger.

If we assume that the initial random estimates are uniformly distributed, the Ordering algorithm converges after $O(\log s)$ steps to a sorted set of nodes, where s denotes the number of rank exchanges, which is bounded by $O(\log n)$.

In the *Ranking* algorithm by Fernández *et al.* [7], each node i picks a set of neighbors uniformly at random, using a background algorithm that can be shown (with simulations) to produce distributions close to uniform. During each gossip round, node i estimates its relative position by comparing the attribute values of its neighbors with its own attribute value. Node i then estimates its rank (and hence its slice) 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 estimate improves.

However, node i does not keep track of the nodes from which it has received values, hence two identical values sent from the same node j are treated by i as coming from two distinct nodes. In part for this reason, while the Ranking protocol achieves eventual accuracy, it can be very slow to converge if the initial selection embodies any form of non-uniformity.

Both of these algorithms require uniform randomness. Unfortunately, neither algorithm is able to guarantee this requirement. As we saw earlier, the key idea of the Ordering algorithm [6] is to use a random number as a rank that is exchanged between nodes. Initially, every node chooses a random number as its rank, then periodically each node compares its value with that of a randomly selected peer. Because the initial random numbers will be used as the final ranks of the nodes, if the number distribution is skewed then final slice estimate can be inaccurate. For example, suppose the initial random numbers of three nodes in a three-node network are 0.1, 0.15, and 0.2 instead of the ideal values 0, 0.5, and 1. When the parallel sort terminates, all three will believe they belong in the first half of the system.

The Ranking protocol suffers from a similar problem associated with the randomness of the initial neighbor selection: if these are biased, the algorithm may converge very slowly.

Churn raises a further potential problem when dynamic events and the attribute values are correlated, for instance if attribute values correspond to average uptimes. The Ordering protocol cannot adjust to variable attribute value distributions, and the Ranking protocol incurs significant delays in its convergence because of this variation.

A. The Sliver Algorithm

This section introduces Sliver, a simple distributed slicing protocol that works by sampling attribute values from the network and estimating the slice number from the sample. More precisely, Sliver temporarily retains the attribute values and the node identifiers that it encounters. With this information, nodes in Sliver make fast and precise estimates of the global attribute value distribution and thus of their correct slice number.

To address churn, we also retain the time at which we last interacted with each node, and gradually age out 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 gossip mechanism 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 can be described as follows.

- Each node i gossips its attribute value to c random nodes in the system.
- Each node j 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.

	Leader-based	Parallel Sorting	Ordering	Ranking	Sliver
Accurate	**yes**	**yes**	no	**yes**	**yes**
Efficient	no	**yes**	**yes**	**yes**	**yes**
Robust to churn	no	no	no	**yes**	**yes**
Handles non-uniformity	**yes**	**yes**	no	no	**yes**
Convergence time	$O(\log n)$	$O(\log^2 n)$	$O(\log s)$	$O(p(1-p)/d^2)$	$O(\sqrt[3]{k^2 \log n})$

TABLE I
COMPARISON OF SOLUTIONS TO THE SLICING PROBLEM. HERE s IS THE NUMBER OF SUCCESSFUL RANK 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, c IS THE CONSTANT NUMBER OF NEIGHBORS, AND k (THE NUMBER OF SLICES) IS AT MOST A CONSTANT FRACTION OF n.

- Each node j sorts the m current values. Suppose B_j of them are lower than or equal to the attribute value of node j.
- Each node j estimates its slice number as the closest integer to kB_j/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 uniformity in the choice of the communicating nodes. Moreover, whereas convergence cannot be established for the Ranking algorithm, Sliver is amenable to a formal analysis.

IV. 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 number using the current distribution of attribute values it has stored. We show analytically that relatively few rounds have to pass for this estimate to be representative for all nodes. More precisely, we derive an analytic upper bound on the expected number of rounds the algorithm needs to run until each node knows its correct slice number (within one) with high probability.

A. Assumptions

We focus on a static 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 c other randomly selected nodes in the system during each round. Clearly, if a node collects all n attribute values it will know its exact slice number. A node i is *stable* if it knows its slice number within at most one. This ensures that nodes whose attribute values lie on the boundary of two partitions are able to converge without having to know most or all values in the system.

B. Convergence to a Sliced Network

In the following we show that Sliver slices the network rapidly. The first lemma gives the number of attribute values a node must receive from distinct nodes in order to be stable. The subsequent theorem

gives the number of rounds necessary to achieve stability with arbitrarily high probability. The corollary is then used to obtain our desired result. Here $\ln(\cdot)$ denotes the natural logarithm with base e.

Lemma 1 *Let $\varepsilon > 0$. For all nodes to be stable with probability $1-\varepsilon$, each node must collect at least*

$$\sqrt[3]{4k^2 \ln\left(1+\frac{2n}{\ln(1/(1-\varepsilon))}\right)}.$$

distinct attribute values.

Proof: Fix some node i with value x_i. We will say that node i receives a previously unknown attribute value in each *time step*. Let B_t denote the number of values that are known after t time steps which are below or equal to x_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 exact slice number. There are on average n/k nodes per slice, so node i is stable as long as it reports a value 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, $B_{t+1} = B_t + 1$, otherwise $B_{t+1} = B_t$. The probability

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

depends on the current distribution of ball colors. Denote this by p_t. To simplify the analysis, we will consider the *Q-process* in which a red ball is picked with probability $q_t = B_n/n$ independently of prior events 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 $p_t \leq q_t$. This means that the P-process tends to move towards B_n/n in each time step, whereas the Q-process ignores the proximity entirely. This means that deviation bounds from B_n/n on the 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. Standard Chernoff-bounds now give

$$\mathbb{P}\left[B_t > \mathbb{E}[B_t] + \frac{t^2}{k}\right] = \mathbb{P}\left[B_t > \left(1+\frac{nt}{kB_n}\right)\mathbb{E}[B_t]\right] < \exp\left(-\frac{\mathbb{E}[B_t](nt)^2}{4(kB_n)^2}\right) = \exp\left(-\frac{tn}{4kB_n}\right) \leq \exp\left(-\frac{t^3}{4k^2}\right)$$

since $B_n \leq n$, and similarly

$$\mathbb{P}\left[B_t < \mathbb{E}[B_t] - \frac{t^2}{k}\right] < \exp\left(-\frac{t^3}{4k^2}\right).$$

Let $s_t = \mathbb{P}\left[B_t > \mathbb{E}[B_t] + \frac{t^2}{k}\right]$ and $s'_t = \mathbb{P}\left[B_t < \mathbb{E}[B_t] - \frac{t^2}{k}\right]$. The probability that all nodes are stable at time t, i.e. B_t/t is within t/k from B_n/n, is at least

$$\prod_{i=1}^{n}(1-s_t)(1-s'_t) \quad > \quad \prod_{i=1}^{n}\left(1-\exp\left(-\frac{t^3}{4k^2}\right)\right)^2 = \left(1-\exp\left(-\frac{t^3}{4k^2}\right)\right)^{2n}.$$

Set $m = 1 - \frac{2n}{\ln(1-\varepsilon)}$ which is clearly $O(n)$ for a fixed value of ε. Now let

$$t \geq \sqrt[3]{4k^2 \ln m}. \tag{1}$$

Then

$$\left(1-\exp\left(-\frac{t^3}{4k^2}\right)\right)^{2n} \geq \left(1-\frac{1}{m}\right)^{2n} = \left(1-\frac{1}{m}\right)^{(m-1)(-\ln(1-\varepsilon))} \geq (1/e)^{-\ln(1-\varepsilon)} = 1-\varepsilon$$

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

The following theorem bounds the expected number of rounds for the system to achieve stability.

Theorem 1 *Let* $\varepsilon > 0$. *After*

$$\frac{n}{c}\ln\left(\frac{n}{n-\sqrt[3]{4k^2\ln\left(1-2n/\ln(1-\varepsilon)\right)}}\right)+O(1)$$

rounds in expectation, all n nodes will be stable with probability at least $1-\varepsilon$.

Proof: Assume for a moment that a node receives only one attribute value per round. The classical *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 $x\ln(x)$. 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, so $\mathbb{E}[T_j] = n/(n-j+1)$. Thus the total time expected to collect t distinct coupons is

$$\sum_{j=1}^{t}\frac{n}{n-j+1} = n(\ln n - \ln(n-t)) + O(1) = n\ln\left(\frac{n}{n-t}\right)+O(1).$$

The $O(1)$ is at most the Euler-Mascheroni constant which is less than 0.6. By plugging in the lower bound (1) of t from the lemma and noticing that each node receives c attribute values per round, the result follows easily. ■

The ln expression in formula (1) in the lemma is deceptive and should be thought of as a weight on n/c between 0 and 1 that depends on the input parameters. The case where the number of slices k exceeds n is uninteresting, so to provide more intuition for the case when k is at most linear (or slightly superlinear) in n we give the following result.

Corollary 1 *If* $k = O\left(\sqrt{n^3/\log n}\right)$, *then all nodes in Sliver will be stable with high probability after* $O(\sqrt[3]{k^2\log(n)})$ *rounds in expectation.*

Proof: Since $k = O\left(\sqrt{n^3/\log n}\right)$, for a fixed $\varepsilon > 0$ there exists a constant α such that after rearranging the equality we get $\sqrt[3]{4k^2\ln\left(1-2n/\ln(1-\varepsilon)\right)} \leq \alpha n$ for large n. Let t denote the expression on the left hand side. Using the theorem, we expect to reach stability with high probability after $\frac{n}{c}\ln\frac{n}{n-t}+O(1)$ rounds. 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

$$n\ln\frac{n}{n-t} = t\left(\frac{n}{t}\log\frac{1}{1-\frac{t}{n}}\right) \leq \frac{t}{1-\frac{t}{n}} \leq \left(\frac{1}{1-\alpha}\right)t = O(t).$$

Dividing by c gives the result. ■

An important special case is that if k is at most a constant fraction of n, then the protocol is expected to converge in $O(\sqrt[3]{k^2 \log n})$ time.

For example, in a system with $n = 100,000$ nodes and $k = 1,000$ slices where each node gossips to $c = 10$ peers, our bound says that we expect the system to be stable with at least 99.9% confidence ($\varepsilon = 0.001$) within 43.2 rounds. In the next section we will run an experiment with $k = 20$ slices, and $n = 3000$ nodes that each gossip to $c = 20$ other nodes. According to our analysis, if all nodes are present and there is no churn then with at least 99.9% confidence we expect the system to become stable within only 3 rounds. These assumptions are rather strong, so to highlight the performance of the algorithm in practice, we use real-life churn in the experiment.

C. Comparing Sliver with Previous Solutions

Here, we compare the performance of Sliver with the performance of other solutions, including the parallel sorting schemes presented in Section VI. Even though Sliver does not compete in terms of convergence with all existing algorithms, it seems to be the best suited approach to solve the distributed slicing in large-scale and dynamic systems. As explained previously, there are several solutions with differing parameters such as convergence-time, accuracy, churn-robustness, and uniformity-requirement. Table I summarizes the characteristics of those solutions. The first column, labeled *Leader-based*, is included for completeness: it summarizes properties of slicing protocols that collect the n attribute values at some single location (a leader), sort them, and distribute k slice boundaries. This would be a reasonable option if n were small, but as the system scales up, the leader quickly overloads.

We see the simplicity of Sliver as a strong advantage. The protocol is gossip-based and does not require that we build any special sort-exchange network. Node arrivals and node departures do not trigger any special work. Sliver is communication-efficient because each node simply sends small messages at a fixed rate to a small number of neighbors, independent of system size. As previously stated in Section IV-B the convergence-time of Sliver is $O(\sqrt[3]{k^2 \log n})$ when k is at most a constant fraction of n. We can expect this complexity to be competitive with that of the Ordering algorithm even though the convergence of this latter algorithm has not been precisely analyzed yet.

The parallel sorting algorithms that will be reviewed in Section VI achieve faster convergence than Sliver. However, as we will see later, these protocols bring a great deal of complexity, and require helper data structures that may need to be reconstructed in the event of churn. In light of this, the Ranking algorithm appears to be the most competitive of the existing options. In order to obtain a fair comparison of these two algorithms, we evaluate both algorithms experimentally in the next section.

V. Experimental Analysis

This section presents an experimental analysis of Sliver. First, Sliver and the Ranking protocols are compared using a real trace of storage space and 90 machines of the Emulab [8] testbed. Then to evaluate scalability, we simulated Sliver on thousands of nodes, using a realistic trace that embodies substantial churn.

A. Distributed Experimentation

We ran an experiment on 90 machines of the Utah Emulab Cluster running 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 [9], 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 4000 machines [10]. The source code of the current experiment and the outputs are available online at `http://lpd.epfl.ch/gramoli/podc08/podc_src.zip` .

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; these are discovered by random walks where special packets are randomly forwarded between nodes in the network for a few rounds, and the requested information is subsequentially sent back to the initiator [11].

Figure 1 compares the performance of Sliver and the Ranking protocol [7] in the settings mentioned above with a timeout of 30 minutes and $c = 1$. The curves represent the evolution of the relative 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 it belongs to by using this position estimate, since it knows the total number of slices k.

At the beginning of the experiment, all nodes have their relative 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 relative position estimate. As a result, even at 1700 seconds, no nodes know their exact position with the Ranking protocol and 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 relative position is exploitable by a node to determine its slice. Depending on the portions of the network the slices represent, the relative position requires to be approximated more or less precisely. For instance, if slices represent small portions (i.e., the number of slices k is large), then the relative position must be precisely approximated. In contrast, if slices represent large portions (i.e., k is low), then a rough approximation of the relative position suffices to determine the right slice.

To better understand the impact of approximating the relative position on determining the slices, we illustrate how fast nodes using Sliver determine their slice number compared to the Ranking protocol while varying the number of slices k. Figure 2 indicates the *slice disorder measure* [7] of the system over time. This disorder is measured as the sum over all nodes of the distance between the correct slice and the slice to which it believes it belongs. More precisely, the slice disorder at time t is $\sum_{i \in A_t} |s_i - e_i|$ where A_t is the set of active nodes at time t, s_i and e_i are respectively the correct slice number and the estimated slice number of node i at time t. The first observation is that in both protocols the convergence slows down as the number of slices enlarges. As mentioned previously, if k enlarges then the portion represented by each slice shrinks; this requires a finer approximation of the relative positions, hence a longer execution of the protocols. 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 $k = n$.

B. 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 [3], using data that was assembled by Guha, Daswani, and Jain [12]. The trace tracks the availability of 3000 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 $n = 3000$ 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 5000 seconds are discarded. The top curve in Figure 3 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 number off by one (or more), and the network quickly converges to have very low slice disorder.

Figures 4 and 5 illustrate the sensitivity of convergence time to k, the number of slices; the results are within the analytic upper bounds derived in Section IV. 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 5000 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.

VI. Prior Work: Parallel Sorting

Earlier, we noted that parallel sorting algorithms can be adapted for use in P2P settings. In this section we review the required steps, undertake a rough complexity estimate, and then review the best known sorting algorithms. We are not aware of any prior investigation of this option.

The structure of this subsection is as follows:

1) We lay out the roadmap by showing that, given a parallel sorting algorithm, slicing can be solved accurately with a single additional phase of message exchanges.
2) Next, we show that an overlay graph can be constructed in a P2P network in a way that will satisfy assumptions typical of parallel sorting algorithms in time $O(\log n)$. In particular, this involves counting the nodes, assigning them sequential identifiers, and creating a routing capability so that nodes can efficiently exchange messages.
3) We review the state of the art in parallel sorting, identifying several best-of-breed solutions, and giving their complexity.
4) Finally, we discuss the implications of churn.

A. Roadmap

Parallel sorting algorithms typically make assumptions that would not hold for the P2P networks in which slicing is desired:

1) These algorithms typically assume that the number of nodes, n, is known in advance, and that nodes have sequential identifiers $0, \ldots, n-1$. In P2P settings the number of nodes is usually not known a-priori and can change as nodes join and leave, or fail. Nodes are generally identified by IP addresses.
2) They typically require that nodes be able to perform pairwise comparison and exchange operations: at least one needs the IP address of the other. In a P2P network, however, the typical assumption

is that each node knows a small constant number of neighbors, and that the graph of connectivity is transitively closed.

As a first step, we show how to take a P2P system satisfying the assumptions used in Sliver and build an overlay network that can resolve all of these requirements. We can then run a parallel exchange sort algorithm on the overlay. In particular, if we sort the attributes of interest, at the outcome of the sorting process, node i (in the sequential numbering) will hold the attribute with rank i in the sort order.

To obtain slice numbers, we simply sort pairs *(attr, id)*: attribute values and these new node numbers, using node-number to break ties if two nodes have the same attribute value. At the end of the sorting algorithm, the i^{th} node will hold a pair $(attr, j)$ that originated on node j; i can then send j a message telling it that its attribute value ended up having index i and hence falls into slice i/k. Thus, with a single additional round, parallel sorting solves slicing.

B. Overlay Construction

Accordingly, we start by developing a very simple P2P protocol that counts and numbers our nodes and constructs an overlay within which they can communicate with one-another. In fact, this is not a hard problem and one can imagine many ways of solving it. For clarity we present a specific protocol and provide a rough analysis.

Any node starts the protocol.

1) Node x sends each of its neighbors a *span(x)* packet containing its IP address.
2) If node y receives *span(x)* and $y < x$ and $root_y < x$ (comparing IP addresses) then y executes step 1, starting its own *span(y)* protocol.
3) Each node y tracks its $root_y$ its $parent_y$ and its $children_y$. Initially, y sets its $root = y, parent = \emptyset, children = \emptyset$. When y receives a *span(x)* message from a neighbor z, if $x \geq root_y$, y *rejects* the message. Otherwise, y sets $root = x, parent = z, children = \emptyset$ and *accepts* the message. y sends an *accept(x)* or *reject(x)* message to z, as appropriate. Then y forwards *span(x)* to all neighbors other than z.
4) If node y receives *accept(x)* from z, it adds z to $children_y$.

If nodes have fanout f it is easy to see that this protocol constructs a spanning tree in time $O(\log_f n)$ rooted at the node with the smallest IP address. Define a *leaf* node for *span(x)* to be a node y that is not x, has $root_y = x$, and receives *reject(x)* from all neighbors other than $parent_y$.

We can use our spanning tree to count n: each leaf node y sends *count(x, 1)* to $parent_y$. An inner node z ignores a $count(x,c)$ message if $root_z \neq x$, and otherwise aggregates the counts for each node in $children_z$, and sends $count(x, sum_z)$ to $parent_z$. The root, x, thus learns the current number of nodes, n, and also how many reside below each of its children. It sends a message *number(x, a)* to each child. An inner node z, receiving *number(x, a)*, ignores the message if $root_z \neq x$, and otherwise sets its own node-id to a and sends number messages to its own children, in the obvious manner.

Thus, in time $O(\log n)$ we construct a spanning tree, compute n, and number the nodes $0, \ldots, n-1$. A leaf node at distance $\log n$ from the root may receive as many as $\log n$ *span* messages, hence the message complexity as seen by the network is roughly $O(n \log n)$. For our purposes below, this rough analysis will suffice.

C. Churn

In the event that some node joins or leaves the system, or fails, it is necessary to rebuild the spanning tree: such events can disconnect the tree or disrupt the numbering, and the parallel sorting algorithms of

interest here are exquisitely sensitive to node numbering and connectivity. However, because our goal is a continuous slicing algorithm that adapts as events occur, there is a simple expedient for dealing with such events. Suppose that we extend the basic protocol with an epoch number in the obvious way, and make the rule that a spanning tree for epoch e' will replace one for epoch e if $e' > e$.

Any node that senses the failure of any other node can simply launch a new instance of the *span* protocol, with a incremented epoch number. A sorting algorithm running on a spanning tree that changes before the sort terminates is simply abandoned mid-way.

Notice that because each membership event can trigger an $O(\log n)$ time overlay construction protocol, there will be a scale of system beyond which this entire method breaks down: for sufficiently large n, the likelihood is that a further churn event would arise before we have time to compute even a single slice. This scaling limit is a further argument in favor of our Sliver protocol, which has no such issue.

D. Parallel Sorting

Most parallel sorting algorithms [13], [14], [15], [16], [17], [18], [19] "wire together" nodes into a sort-exchange network, within which they can compare their value and decide whether to exchange their position. Our distributed overlay structure permits us to run such algorithms in P2P settings: for node i to compare and exchange its value with node j, if i and j have not previously communicated, we can route a message from one to the other within the overlay, for example from i to j through some common ancestor. When j receives this message, it learns the IP address of i and can reply directly, at which point i learns the IP address of j. This brings an overhead: the first message will take $O(\log n)$ hops, and the root node may need to forward a high percentage of such messages.

As noted earlier, our overlay construction approach makes sense only if the overlay is used *for an extended period of time* before being reconstructed. The algorithms discussed below use sort-exchange networks in which the same comparison links are employed for each instance of the sorting problem. Thus, the $O(\log n)$ cost of constructing shortcut links will only be incurred once, after which the shortcuts will be used many times. Recall that the cost of building the overlay itself was $O(\log n)$. Thus the delay of first-time routing will not impact the predicted convergence time of the parallel sorting algorithms we consider.

We should again note that there are many ways to build spanning trees, count nodes, number nodes and build routing structures. The scheme we've outlined is offered simply as an example of a fast, efficient way to adapt parallel sorting to the task at hand without increasing the time complexity of the underlying sorting algorithm.

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 [20]; nonetheless, it remains over 1000. Batcher's algorithm [13] has complexity $O(\log^2 n)$. Although an $O(\log n \log\log n)$-step algorithm is known [17], 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 [17] 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 her rank to good accuracy.

We can thus expect a probabilistic parallel sorting algorithm to solve slicing exactly within time $O(\log n)$: independent of k and hence asymptotically faster than what we can achieve with Sliver. Nonetheless, we believe that our gossip-based solution is preferable: the construction is simpler and the protocol is reasonably efficient.

VII. Conclusion

Our paper evaluates a number of possible solutions to the distributed slicing problem. We introduce a new protocol, Sliver, and offer an analysis of its convergence properties. Some prior protocols converge slowly, some don't guarantee accuracy, and some (the adaptations of parallel sort) 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] S. Saroiu, P. K. Gummadi, and S. D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proc. of Multimedia Computing and Networking*, 2002.

[2] D. Stutzbach and R. Rejaie, "Understanding churn in peer-to-peer networks," in *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*, 2006, pp. 189–202.

[3] "Skype homepage," http://www.skype.com .

[4] "Gnutella homepage," http://www.gnutella.com .

[5] "KaZaa homepage," http://www.kazaa.com .

[6] M. Jelasity and A.-M. Kermarrec, "Ordered slicing of very large-scale overlay networks," in *Proc. of the Sixth IEEE Conference on Peer-to-Peer Computing*, 2006.

[7] 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.

[8] 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 *OSDI02*, 2002, pp. 255–270.

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

[10] 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*, 1999, pp. 59–70.

[11] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 8, 2007.

[12] 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.

[13] K. Batcher, "Sorting networks and their applications," in *AFIPS 1968 Sprint Joint Comput. Conf.*, vol. 32, 1968, pp. 307–314.

[14] M. Ajtai, J. Komlós, and E. Szemerédi, "Sorting in c log n parallel steps," *Journal Combinatorica*, vol. 3, no. 1, pp. 1–19, March 1983.

[15] F. M. M. auf der Heide and A. Wigderson, "The complexity of parallel sorting," *SIAM J. Comput.*, vol. 16, no. 1, pp. 100–107, 1987.

[16] R. Cole, "Parallel merge sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770–785, 1988.

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

[18] 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*. New York, NY, USA: ACM Press, 1997, pp. 121–128.

[19] M. T. Goodrich, "Communication-efficient parallel sorting," *SIAM J. Comput.*, vol. 29, no. 2, pp. 416–432, 1999.

[20] V. Chvátal, "Lecture notes on the new AKS sorting network," Rutgers University, Tech. Rep., 1992.

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

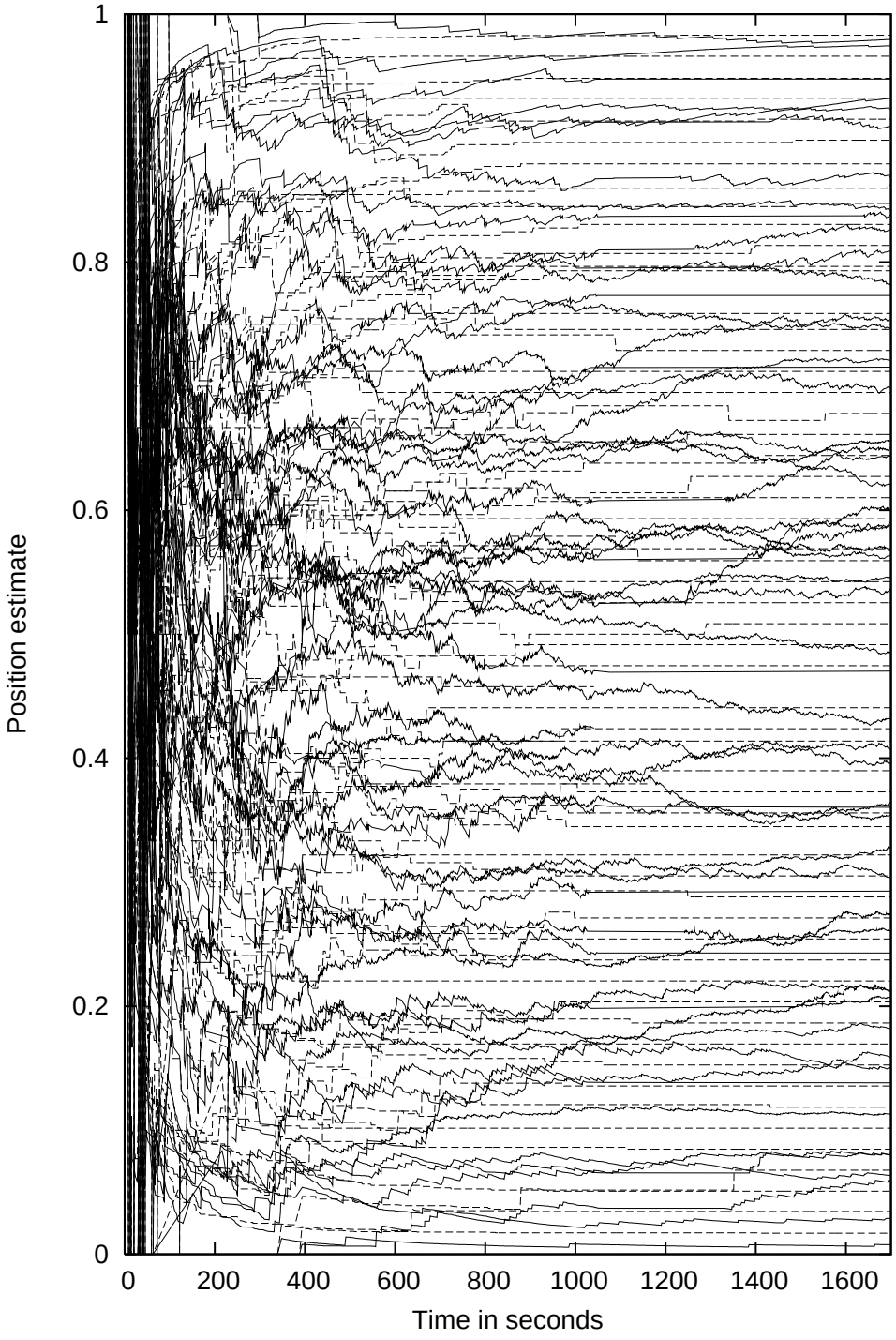

Fig. 2. Convergence time depending on the number of slices k.

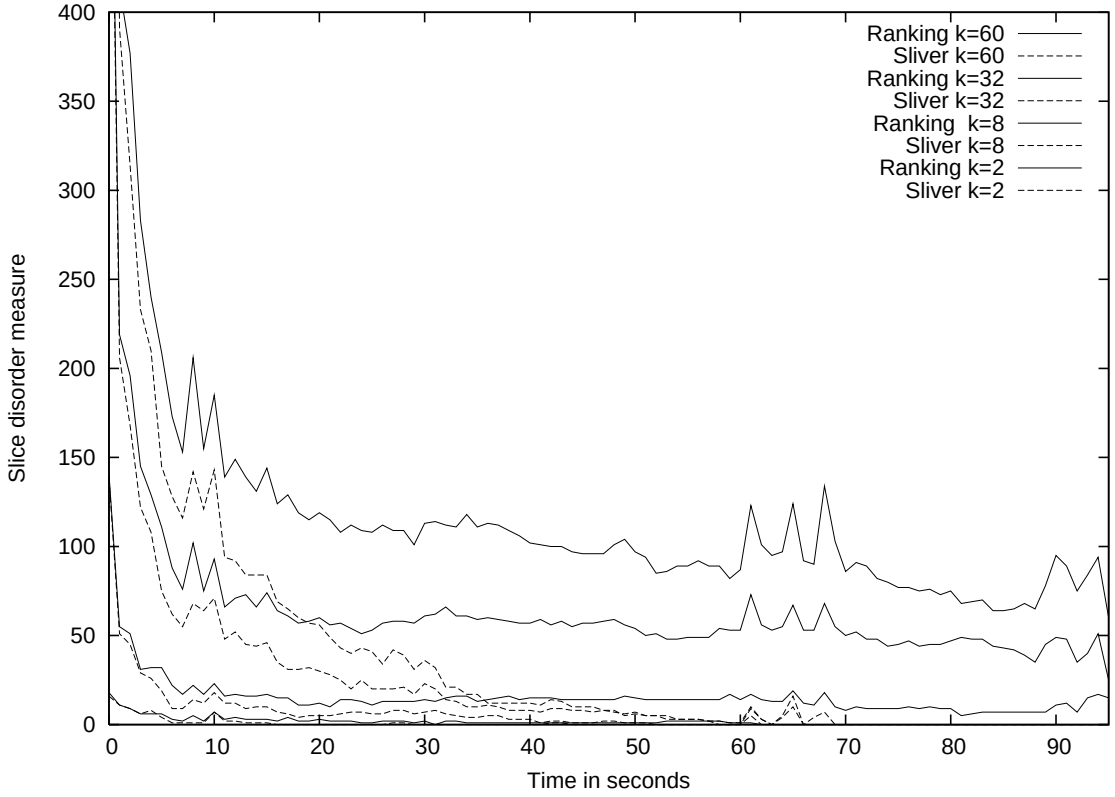

Fig. 3. Slice disorder measure (bottom) of Sliver along with number of active nodes (top) in a trace of 3000 Skype peers.

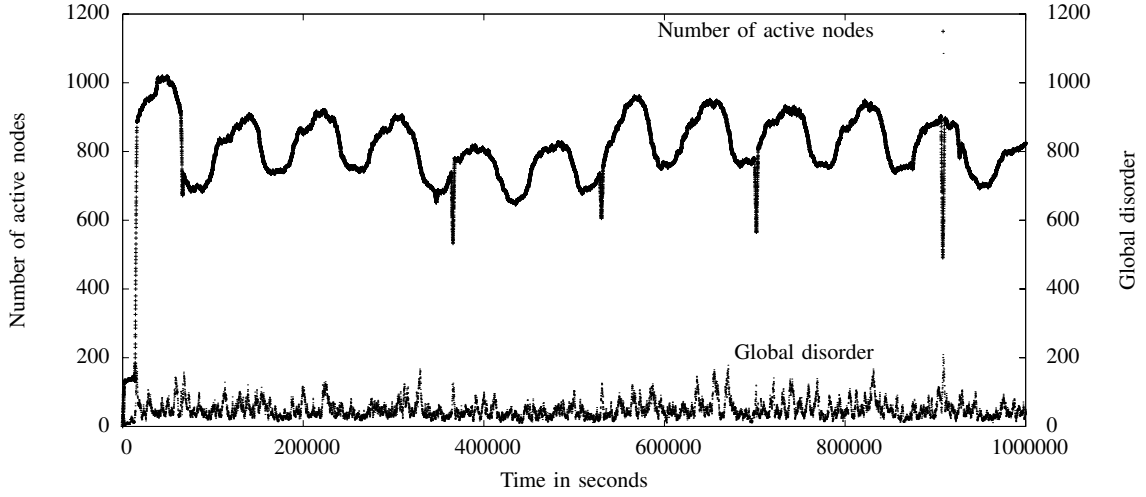

Fig. 4. Average slice disorder and number of misreporting nodes over the first 100,000 seconds in the Skype trace of 3000 nodes as a function of the number of slices. Error bars represent one standard deviation from the mean, and are drawn for every three data points for clarity.

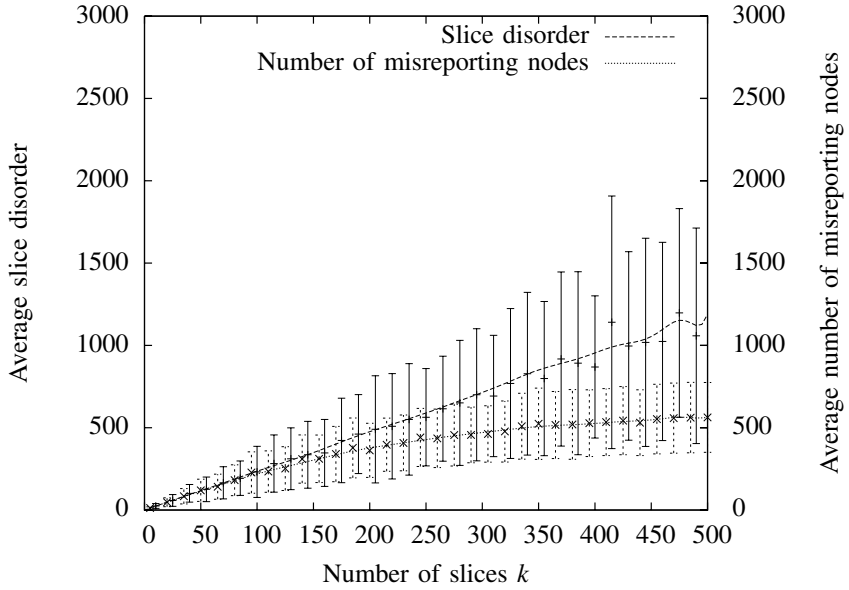

Fig. 5. Fraction of misreporting nodes at every step averaged over the first 100,000 seconds in the Skype trace as a function of the number of slices. Error bars represent one standard deviation from the mean, and are drawn for every three data points for clarity.