Slead: low-memory, steady distributed systems slicing

Francisco Maia[1], Miguel Matos[1], Etienne Rivière[2], and Rui Oliveira[1] *

[1] High-Assurance Software Laboratory, INESC TEC & University of Minho, Portugal
{fmaia,miguelmatos,rco}@di.uminho.pt
[2] Université de Neuchâtel, Switzerland.
etienne.riviere@unine.ch

Abstract. Slicing a large-scale distributed system is the process of autonomously partitioning its nodes into k groups, named *slices*. Slicing is associated to an order on node-specific criteria, such as available storage, uptime, or bandwidth. Each slice corresponds to the nodes between two quantiles in a virtual ranking according to the criteria.
For instance, a system can be split in three groups, one with nodes with the lowest uptimes, one with nodes with the highest uptimes, and one in the middle. Such a partitioning can be used by applications to assign different tasks to different groups of nodes, e.g., assigning critical tasks to the more powerful or stable nodes and less critical tasks to other slices.
Assigning a slice to each node in a large-scale distributed system, where no global knowledge of nodes' criteria exists, is not trivial. Recently, much research effort was dedicated to guaranteeing a fast and correct convergence in comparison to a global sort of the nodes.
Unfortunately, state-of-the-art slicing protocols exhibit flaws that preclude their application in real scenarios, in particular with respect to cost and stability. In this paper, we identify steadiness issues where nodes in a slice border constantly exchange slice and large memory requirements for adequate convergence, and provide practical solutions for the two. Our solutions are generic and can be applied to two different state-of-the-art slicing protocols with little effort and while preserving the desirable properties of each. The effectiveness of the proposed solutions is extensively studied in several simulated experiments.

1 Introduction

Current information systems are being deluged by sheer amounts of data that need to be processed and managed [7]. At the same time, processors are not getting faster at the same rate of previous years but instead it is possible to

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-015020 and EU FP7 project CumuloNimbo: Highly Scalable Transactional Multi-Tier PaaS (FP7-257993).

have more of them [16] making it possible to consider thousands of machines, each with hundreds of processors, in alternative to more expensive and centralized architectures. Taking advantage of such massive scale deployments requires the design of suitable protocols. In particular, epidemic or gossip-based protocols have been successfully used to address a multitude of problems from data dissemination, decentralized management, data aggregation or publish/subscribe [15].

A typical epidemic protocol operates as follows. Each node has some locally produced/gathered knowledge and a set of neighbors, called its *view*. The protocol progresses by having each node periodically and continuously exchange knowledge with one or several of its neighbors, each partner of the exchange then updating its local state.

Large-scale systems are usually composed of highly heterogeneous nodes, according to their capacity, stability or any other application-specific requirements. The ability to distinguish between groups of nodes based on a discrete metric reflecting a criteria, allows to dynamically provision nodes to certain tasks according to their desirability. For instance, nodes with a higher uptime tend to be more stable for a given additional period than those with a small uptime [2]. Partitioning the set of nodes into k several groups of increasing uptime, allows to assign critical services to more stable nodes, and less critical services to less stable ones. Examples include assigning privileged roles to more stable nodes to improve the quality of a streaming application [18], or allocating a data partition to a group of nodes in a key-value store [11]. The operation of partitioning in k groups according to node-specific criteria is called distributed slicing [6, 9, 13].

Slicing is an *autonomous process* by which each node in the system shall decide to which slice it belongs. The decision is intuitively based on a virtual global ranking of all nodes according to the criteria: based on its rank, it is straightforward for a node to decide to which of the k slices it belongs. Obviously, given the scale and dynamics of the systems we consider, it is intractable to locally gather all nodes' characteristics and perform the ranking in one place. The decision needs to be made by each node *individually* in a completely *decentralized* manner, based on the knowledge of its own value, the values of (some) other nodes, and the slicing parameter k. Of course, such a decentralized protocol operating on a large-scale dynamic system is based on compromises between accuracy and convergence speed, reactivity to population changes and costs.

Unfortunately, despite the usefulness of slicing, state-of-the-art protocols still exhibit flaws that preclude, in our opinion, their immediate applicability as building blocks for large-scale applications. In this paper, we analyze these state-of-the-art protocols and focus on three previously disregarded metrics: *steadiness*, *slice variance* and *memory complexity*.

Steadiness is the ability of the protocol to take slice changes decisions only when necessary. It is the opposite of *slice instability*, measured by the distribution of the number of slice changes per second. A slice change can be legitimate, e.g., if the value of the nodes' attributes and thus the virtual ranking change, or if the size of the system changes. However, a slice change typically implies a considerable load for the overlying applications, as it requires reconfiguring the

node for its new role, and often reconfiguring other nodes to take over its previous responsibilities. Undesired slice changes or oscillations between two slices tend to appear more frequently for nodes that lie at the "borders" of slices, that is, at the boundary of slices in the *virtual* ranking of all attributes. For instance, in the key-value store application mentioned above [11], a slice change results in discarding a potentially large fraction of hard state for the current slice and getting the new state from nodes of the new slice, which can be costly.

Slice variance is a metric that reflects the correctness of the nodes allocation to slices, and in particular, the size distribution of the slices. It is important to notice that this metric significantly differs from the slice disorder metric used in previous work [9]. Slice variance does not distinguish whether a specific node is in the correct slice all the time but instead if the overall distribution of nodes into slices is close to the expected one, i.e., each slice is close in size to $\frac{N}{k}$ as possible (N is the size of the system). The *slice variance* is defined as the variance measured between the observed distribution of slices and $\frac{N}{k}$.

Finally, we consider the *memory complexity* imposed on nodes for deciding on their slice. This is a fundamental metric to assess scalability. A linear complexity requires keeping information in the order of the size of the system, and to maintain it through the system's dynamics, leading to poor performance and high costs.

We conducted experiments with two state-of-the-art protocols for distributed slicing [6, 8]. These protocols exhibit reasonable *slice variance* but suffer from serious *steadiness* and *memory complexity* problems. We address the two issues without impairing the original protocols performance w.r.t. other metrics. Our proposal, which we named SLEAD, is a novel distributed slicing protocol whose design principles are generic enough to be adapted to other protocols such as [6, 8]. We address both issues with a *hysteresis* mechanism that significantly enhances *steadiness*. It is coupled with a bounded-memory state management mechanism based on Bloom filters [3] that allows us to control *memory complexity* with a very limited impact on convergence and accuracy.

The remainder of the paper is structured as follows. In Section 2, we present current state-of-the-art protocols and their evaluation according to the metrics above. Section 3 presents our contribution, SLEAD. We conclude and highlight some future work guidelines in Section 4.

2 Distributed Slicing: State-of-the-Art

In this section we present, analyze and discuss two protocols, Ranking [6] and Sliver [8] that to the best of our knowledge represent the state-of-the-art for distributed slicing. A complementary review and comparison of these protocols and other distributed slicing approaches can be found in [9].

In general, each node participating in a slicing protocol possesses an arbitrary local attribute and wishes to know the slice this value belongs to. The protocols work by performing pairwise exchanges of the local attribute with its neighbors. The decided slice may change after each such exchange, when the locally available

information indicates that the local attribute value crosses a border in the global *virtual* ranking.

By assumption, each node in the system has access to a continuous stream of random nodes from the system. These nodes can be used as members of the node's view or to determine its position among the different slices. This is usually provided by an underlying proactive Peer Sampling Service (PSS) [10] that builds this stream of random nodes through a gossip-based periodic exchange of views between nodes. We also assume that the number of slices, k, is known by all nodes. This value can easily be disseminated to all nodes through a gossip-based dissemination [5], leveraging the PSS.

2.1 Ranking

Ranking [6], described by Algorithm 1, works in periodic cycles. It features an active and a passive thread. At each cycle, a node's *active thread* updates the local view by obtaining fresh random peers from the PSS. It then initiates an exchange with all these peers, simply sending its attribute (lines 7 to 10). Each contacted node processes the request with its *passive thread* (lines 11 to 27).

The principle of Ranking is to locally estimate the number of received attributes that are smaller than the receiver's. This allows estimating the position of the node's attribute in the *virtual* ranking, and decide on a slice (line 27). Ties in attribute values are disambiguated by comparing the node identifiers (line 16, second clause of the condition). Failure to do so by considering tied attributes on either the smaller or greater portion of the system would introduce estimation problems, particularly in scenarios where the attribute distribution is narrow (multiple nodes with the same attribute value).

As described, Ranking uses a sliding window mechanism by bounding the number of attributes considered and thus take churn (nodes' dynamics) into account.

2.2 Sliver

Sliver [8], described by Algorithm 2, relies on the same basic idea of Ranking. Its fundamental difference though is to not only keep track of the attributes received but also to record their source nodes. Such apparently small difference has a significant impact and tackles a weakness in Ranking. Because the PSS is proactive and nodes periodically exchange the same information, eventually Ranking will consider the same attributes (providing from the same nodes) several times in the slice computation. If the underlying PSS does not provide completely uniform samples of the network, for instance due to heterogeneous network connections or to the nature of the shuffling operation used,[3] the biasing may strongly affect the accuracy of the slice estimation [9]. The longer the time slice considered,

[3] As demonstrated in [10] there is no such thing as a "perfect" peer sampling service; protocols that favor reactivity to take into account failed nodes usually impose a clustering ratio that is higher than that of a purely random network. It means that nodes in the vicinity of a given node are more likely to be seen twice in the flow of random nodes than what would have been the case with a purely random network.

```
1  initially
       // view provided by the PSS
2      view ← ∅
       // local attribute
3      myAttribute ← ...
       // number slices, system parameter
4      k ← ...
       // list of latest collected attributes
5      attributeList ← ∅
       // current slice estimation
6      slice ← ⊥

   // active thread
7  every Δ sendAttribute()
8      view ← PSS.getView()
9      foreach p ∈ view
10         send myAttribute to p

   // passive thread
11 receive value from p
       // number of smaller attributes seen
12     smaller ← 0
       // total number of attributes seen
13     total ← 0
14     if attributeList.full then
15         attributeList.removeOlder()
16     if (value < myAttribute) ∨
17      (value == myAttribute ∧
18       p < myid) then
19         attributeList.add(true)
20     else
21         attributeList.add(false)
22     foreach a ∈ attributeList
23         if a then
24             smaller ← smaller + 1
25     total ← attributeList.size()
26     position ← smaller / total
27     slice ← k * position
```

Algorithm 1: Ranking [6].

```
1  initially
       // view provided by the PSS
2      view ← ∅
       // local attribute
3      myAttribute ← ...
       // number slices, system parameter
4      k ← ...
       // holds the received attributes and
          node ids
5      attributeList ← ∅
       // current slice estimation
6      slice ← ⊥

   // active thread
7  every Δ sendAttribute()
8      view ← PSS.getView()
9      foreach p ∈ view
10         send myAttribute to p

   // passive thread
11 receive value from p
       // number of smaller attributes seen
12     smaller ← 0
       // total number of attributes seen
13     total ← 0
14     if attributeList.contains(p,value)
       then
           // pair attribute and id become
              the head of list
15         attributeList.update(p,value)
16     else
17         if attributeList.full then
18             attributeList.removeOlder()
19             attributeList.add(p,value)
20         else
21             attributeList.add(p,value)
22     foreach a ∈ attributeList
23         if a.value < myAttribute then
24             smaller ← smaller + 1
25         else
26             if a.value == myAttribute
               ∧ a.id < myId then
27                 smaller ← smaller + 1
28     total ← attributeList.size()
29     position ← smaller / total
30     slice ← k * position
```

Algorithm 2: Sliver [8].

the more important is the bias introduced by selecting the same nodes several times. As Sliver keeps track of nodes identifiers, it is possible to overcome the impact of duplicates as well as provide a convergence proof as shown in [9]. Such a convergence proof is not applicable to Ranking.

2.3 Using a sliding window of observation

Unfortunately, the continuous collection of attributes hinders scalability, as the memory required is proportional to the system size. This is the case for Ranking but is even more critical in Sliver as much more information is kept for each interaction. Due to this, both protocols bound memory usage by defining a *time to live* on attribute records, which enables to adjust memory consumption. In practice, defining a *time to live value* is equivalent to defining a maximum number of records each node can store. In our experiments this is the approach taken by keeping the records in a least-recently-used structure with custom size.

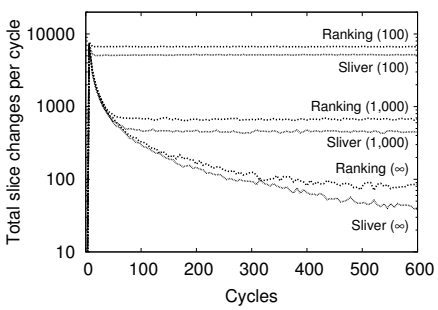
It is important to notice that the ability to forget records is crucial to cope with churn and changes in node local attribute values albeit with an impact on steadiness. In fact, defining a low value for the maximum amount of memory used allows the system to adapt to changes very fast but at the cost of unsteadiness, whereas increasing memory improves stability but slows the response to change.

2.4 Evaluation of Ranking and Sliver

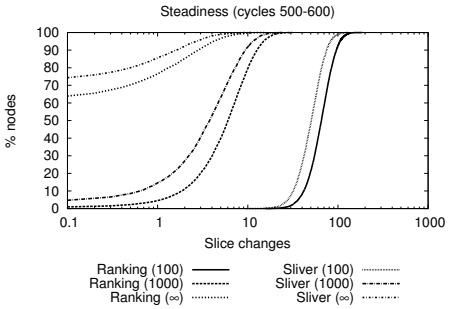
In this section we study the behavior of Ranking and Sliver with respect to *Steadiness* and *Slice variance*, for different amounts of memory consumption. The experiments were conducted with the help of the PeerSim simulation framework [12] with a system size of 10 000 nodes and $k = 10$ slices with the event-based engine. For each experiment both protocols are stacked on top of the same PSS (Cyclon [17] in our case) and thus receive the same views enabling a direct comparison of results. As indicated in [10], Cyclon provides the best results of available PSS for the quality of the randomness of the streams of nodes constructed (in particular, low clustering ratios). This means we consider the best conditions for Ranking here; accuracy can only get worse as other PSS are considered. All presented results are the average of 10 executions. Due to the large number of points to plot, we applied a cubic spline transformation that summarizes plot data in order to improve readability. We consider the following configurations: Ranking and Sliver with memory size (maximum number of elements in *attributeList*) of 100, 1,000 and ∞.

For all configurations, the size of the view is 20. This means that the active thread of both Ranking and Sliver will contact 20 nodes with their attribute value. If we consider the network formed by the PSS views to be random (a reasonable assumption in this case), each node will be on average contacted 20 times per cycle. Every time a node is contacted with an attribute value, its passive thread will integrate the received value and may decide on a slice change. In the worst case, a node may thus change its slice 20 times per cycle.

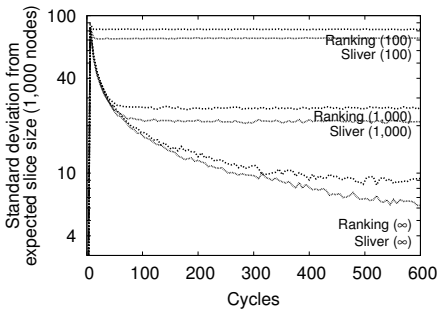
Figure 1(a) explores the *steadiness* of the various configurations. We represent the evolution of the number of changes per cycle, for all nodes (note the



(a) **Steadiness.** Evolution of the number of slice changes.



(b) **Steadiness.** Cumulative changes over the last 100 cycles.



(c) **Slice Variance.** Evolution of the slices std. dev. from 1,000 nodes.

Fig. 1. *Steadiness* and *slice variance* for 10,000 nodes and 10 slices over 600 cycles.

logarithmic scale for the y axis). As expected, due to the low number of values stored by both protocols, there is a major instability of the slice decisions in the beginning that result in a large number of slice changes, multiple times per cycle and per node.When using a bounded memory size, there is a *stabilization period* after which the number of slice changes per cycle remain almost constant. This stabilization period is the time it takes to fill the memory: 20 times 50 cycles makes for 1,000 entries in one case, 20 times 5 cycles makes for the 100 entries in the other. The number of slice changes, and thus *steadiness*, is thus directly linked to the memory size at each node.

Even a memory of *a tenth* of the total system size is synonym with major slice attribution instability. Keeping system-size amount of information results in the protocols stabilizing, but very slowly. By cycle 600, Ranking will have seen 600 times 20 values, more than the size of the system, and still be unstable. As expected, Sliver is slightly more efficient for the same memory and stabilizes faster by discarding already known information and counting each attribute only once. Nonetheless, we do not see the stabilization of Sliver with a complete knowledge of the system as it would require much more than $\frac{10{,}000}{20} = 500$ cycles

to get such a complete knowledge (latest missing attributes taking longer to be captured). We note that the difference between Ranking and Sliver would be higher if using a PSS yielding a lower-quality stream of nodes, e.g., where clustering would be more present.

Figure 1(b) presents the cumulative slice changes from cycle 500 to 600 which is enough for all configurations to stabilize. As expected, slice changes are not evenly distributed among all nodes and tends to affect nodes that are on, or next to, *slice borders* in the virtual ranking. In fact, even with knowledge of one *tenth* of the system (1 000 records), roughly 20% of the nodes change slices at least every 10 cycles. The result is deceptive for the usability of Ranking and Sliver in a real system as these nodes will be unusable or incur a heavy and persistent reconfiguration load on the system.

Figure 1(c) presents the impact of the various configurations on *slice variance*. Here, we plot the standard deviation from the expected slice size (1,000 nodes). We observe that *slice variance* is heavily dependent on the memory used: more entries reduce the differences between slices while low memory (100 entries) results in an instability on the number of slices. Note that the distribution of slice sizes evolves over time: the large slices may be the smaller a few cycles later, due to the randomness in the slice attribution. This we attribute to the low memory available and resulting limited knowledge of the network.

Discussion These evaluations show that an immediate application of either protocol is problematic, particularly due to the *steadiness* problem, as a significant percentage of the system would be devoted to performing slice transitions without doing any useful work. These observations are the starting point and main motivation behind the solutions and protocol presented next.

3 Slead

In this Section we present Slead, a new distributed slicing protocol that addresses the problems of *steadiness* and *memory consumption* found in existing protocols and highlighted in the previous section. This is achieved without impacting *slice variance* (and thus the distance from an ideal slice distribution). In fact, Slead can achieve the same *slice variance* as state-of-the-art protocols but with a significantly lower memory consumption as we demonstrate later in this Section. For the sake of clarity we introduce each mechanism independently which allows a better understanding of the impact of each of them.

Conceptually, Slead is similar to both Sliver and Ranking as in each cycle nodes send their local attributes to their neighbors and compute their position in the global ranking (and hence their slice) based on the attributes received in the recent past. The full pseudo-code of Slead is presented in Algorithm 3, and detailed and evaluated in the following sections.

3.1 Steadiness

Changing slice typically requires the node to change context and local state, which can be very expensive. As we have shown in Section 2, Sliver [8] and Ranking [6] suffer from a *steadiness* problem in the slice estimation: a large

fraction of nodes keep changing slices even in a stable network and long after bootstrap. In fact, this happens mainly because nodes close to the slice border are highly affected by small variations in their position estimation.

To address such fluctuations, we propose the use of a *hysteresis* mechanism that prevents such problematic changes. The basic idea is to look at the slice estimate over a period of time and only change slice if the slice proposal is done for a sufficient amount of rounds, or if the magnitude of the change is high e nough. The number of rounds or the magnitude of the change needed is given by a parameter we call the *friction factor*.

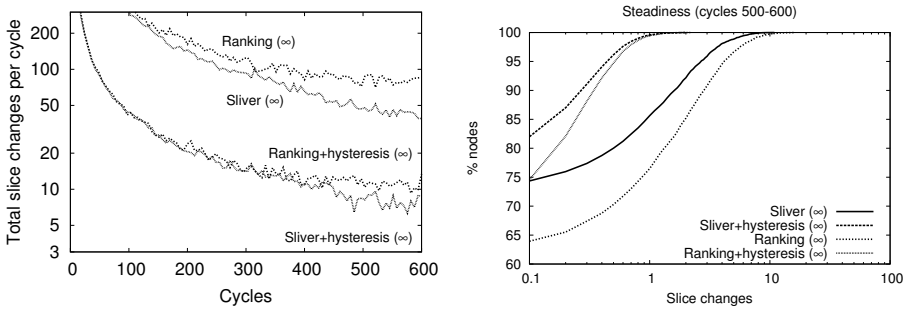
The hysteresis component of SLEAD is presented in Algorithm 3, lines 20 to 24 and works as follows. At each cycle, the protocol computes the slice estimation (lines 18 to 20). The magnitude of the change is accumulated in a local variable, *current_difference*, which represents the cumulative difference between the current slice estimation and the one the protocol is suggesting as correct (line 21). As we compute the difference between the current slice and the estimated one, small fluctuations in the estimation are avoided since they do not go over the *friction factor* and thus *steadiness* is improved. If the estimated slice consistently points to a new value, the cumulative difference will eventually be greater than the *friction factor* and the protocol will effectively adopt the change to the new slice. Furthermore, as the hysteresis is based on cumulative differences the protocol is able to quickly adapt to abrupt changes in the system such as massive joins or failures. In fact, if the difference between the proposed slice and the current one is greater than the *friction factor*, the change will be immediate thus helping to effectively deal with dynamics.

Figure 2 presents the impact of the hysteresis mechanism applied to Ranking and Sliver in the same scenario of Section 2 with *friction*=2. We only consider the versions with unbounded memory of both protocols as those achieve better results in both metrics as observed in Figure 1. We observe that the hysteresis mechanism not only improves overall system *steadiness* (Figure 2(a)) but also considerably reduces the amount of nodes that frequently changes slice (Figure 2(b), note that the x axis scale is logarithmic). Moreover, there is no impact on *slice variance* (Figure 2(c)) meaning that despite avoiding unnecessary changes the protocols still converge to the optimal configuration when compared with their original versions.

3.2 Memory usage

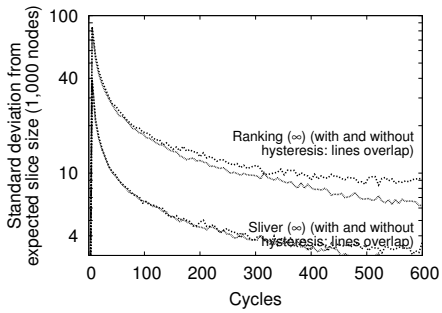
The other main frailty with existing slicing protocols is that the memory requirements depend on the system size and too low a memory impacts *slice variance* as observed in Figure 1. This is because Ranking and Sliver need to store the values of the attributes of other nodes (and the node *id* in the case of Sliver) to build adequate estimations of the slice position. The compromise taken in Sliver and Ranking is to use a least-recently-used structure that bounds memory consumption even though constraining estimation accuracy.

Our contribution to reducing memory usage rests on two key observations regarding the nature of distributed slicing. First, it is important to track which attributes (source nodes) have been considered in the past to avoid duplicates.



(a) **Steadiness.** Evolution of the number of slice changes.

(b) **Steadiness.** Cumulative changes over the last 100 cycles.



(c) **Slice Variance.** Evolution of the slices std. dev. from 1,000 nodes.

Fig. 2. Impact of hysteresis on *steadiness* and *slice variance* (10,000 nodes, 10 slices).

Secondly, what really matters to the slice computation is not the values themselves but whether they are greater or smaller than the local attribute. The first observation directly calls for the use of a Bloom filter, a space-efficient data structure for tracking identifiers [3][4]. The second one, leads to simply counting the greater and smaller observations, which only requires to keep two numbers instead of a list with all the occurrences.

Therefore, in SLEAD we use Bloom filters to track the node identifiers, which allows to track a significant higher number of ids using a bounded and small amount of memory. Assuming a pair `IP:port` as the node identifier (48 bits) and that attributes are encoded as long integers (64 bits), each entry requires 64 bits in Ranking and 112 in Sliver. For the memory configurations used previously with 100, 1000 and 10,000 entries (the unbounded version in practice corresponds to the system size), Ranking requires 6,400, 64,000 and 640,000 bits, whereas Sliver requires 11,200, 112,000 and 1,120,000 bits, respectively. On the other

[4] We note that using a Bloom filter can give false positives for the inclusion of an element in the set (here, a node identifier). However, the probability of a false positive for the identifier of a node with a greater attribute is the same as for a node with a smaller attribute; henceforth the position estimation is not affected by such errors that are evenly spread on the attribute range space.

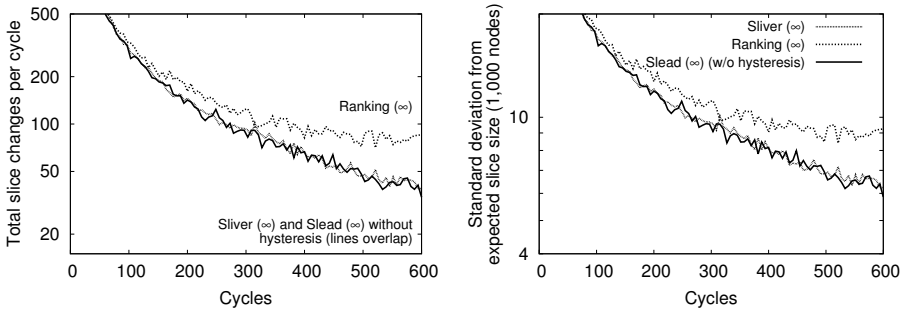
hand, a Bloom filter with a probability of false positives of 1×10^{-4} (the order of the system size) requires only 1,071, 10,899 and 109,158 bits for storing 100, 1,000 and 10,000 nodes respectively [3], representing savings of around 90% when compared to Sliver. The next step is simply to count the number of elements in each Bloom filter and compute the slice estimation accordingly (lines 10 and 19). Please note that the addition to a Bloom filter is an idempotent operation and thus has no impact on the cardinality which can be easily computed from the filter fill ratio [3].

```
1  initially
       // view provided by the PSS
2      view ← ∅
       // local attribute
3      myAttribute ← ...
       // number slices, system parameter
4      k ← ...
       // node identifiers whose attributes are smaller than the local one
5      smaller ← BloomFilter()
       // node identifiers whose attributes are greater than the local one
6      greater ← BloomFilter()
       // current slice estimation
7      slice ← ⊥
       // current value of cumulative changes attempts
8      current_difference ← 0

   // active thread
9  every Δ sendAttribute()
10     view ← PSS.getView()
11     foreach p ∈ view
12         send myAttribute to p

13 receive value from p
14     if (value < myAttribute ∨ (value == myAttribute ∧ p < myId)) then
15         smaller.add(p)
16     else
17         greater.add(p)
18     total ← smaller.size() + greater.size()
19     position ← smaller.size() / total
       // hysteresis mechanism
20     nextSlice ← k * position
21     current_difference ← current_difference + (slice − nextSlice)
22     if ‖current_difference‖ > friction then
23         slice ← nextSlice
24         myprotocol.current_difference ← 0
```

Algorithm 3: SLEAD protocol.

To evaluate our mechanism, we compared Ranking and Sliver with unbounded memory which in practice corresponds to 640,000 and 1,120,000 bits respectively, and SLEAD with 218,316 bits which corresponds to the two Bloom filters with a capacity to store 10 000 node identifiers with a false positive probability of 1×10^{-4}. We detail the need for two bloom filters in the next section. To isolate the impact of the use of Bloom filters, SLEAD does not use the hysteresis mechanism in this experiment. The results are depicted in Figure 3 and as it is possible to observe despite using only 35% of Ranking's memory and 20% of Sliver's, SLEAD provides similar results for both *steadiness* and *slice variance*. Such memory improvements could be further increased by using more advanced

Bloom filters that do not require setting an a priori filter size and are able to scale with the number of inserted elements [1]. In fact, this benefits nodes that are on the low/high end of the attribute spectrum as they will not require significant memory for the smaller/larger Bloom filters.



(a) **Steadiness.** Evolution of the number of slice changes.

(b) **Slice Variance.** Evolution of the slices std. dev. from 1,000 nodes.

Fig. 3. Bloom filter's impact on *steadiness* and *slice variance* (10,000 nodes, 10 slices).

3.3 Dynamics

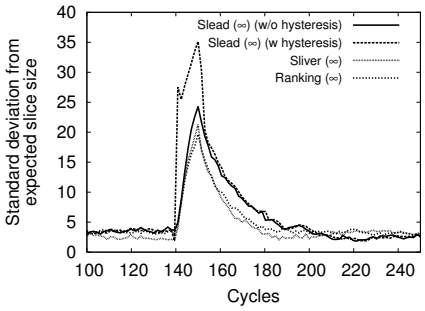
In the previous section we intentionally omitted details regarding the Bloom filter implementation. Actually, such implementation impacts the behavior of the protocol, which can be tuned to meet application specific criteria.

A traditional Bloom filter implementation [3] does not have the ability to delete entries. In the static scenarios we considered previously such capacity is not required and moreover, due to the low memory consumption, this simple Bloom filter implementation copes with our requirements. However, in scenarios with churn this capacity is fundamental as it enables old values to be pruned enabling adaption to new configurations. In Ranking and Sliver this is addressed by the sliding window mechanism, which simultaneously limits memory usage.

In Slead we decouple these distinct but related properties simply by considering a different implementation of the underlying Bloom filter. To this end we use an implementation able to forget and mimic the sliding window-type behavior found in Ranking and Sliver. The approach used, known as A^2, provides least-recently-used semantics while keeping low memory usage [19]. In short it uses two traditional Bloom filters that are filled out of phase, i.e. one starts to be filled only after a number of updates to the other. This allows each Bloom filter to record a set of values that differ in the timeline they represent, where one contains the more recent items and is a subset of the other. The old values are deleted by judiciously swapping and flushing the Bloom filters [19].

In our experiments we used the A^2 implementation with the parametrized memory size. Figure 4 presents the evaluation of Slead under a dynamic environment and thus the impact of A^2. We start with a system with 100 nodes, let it stabilize, and then at cycle 140 add 10 nodes per cycle for a duration of 10 cycles. As it is possible to observe, Slead exhibits similar behavior to Sliver

and Ranking. Even though it incurs in slightly higher variance initially, it quickly converges and accommodates the system size changes. Moreover, when the hysteresis mechanism is added, the same quick convergence is observable validating that our complete approach is also adequate for dynamic environments.



(a) **Slice Variance:** evolution of the slices std. dev.

Fig. 4. Slice variance under churn. Starts with 100 nodes, ends with 200.

4 Discussion

In this paper we studied the behavior of two state-of-the-art distributed slicing protocols, Ranking and Sliver, along several practical metrics namely, *steadiness*, *slice variance* and *memory complexity*.

The experiments conducted showed that acceptable *slice variance* could only be achieved with considerable memory consumption which poses inherent scalability limits. Moreover, memory usage also impacts *steadiness* which imposes constant slice reconfigurations. For instance, even keeping track of one tenth of the node identifiers in the system, more than 10% of the nodes keep changing slice very frequently and thus cannot be used effectively (Figure 1).

Our proposal, SLEAD, overcomes these limitations by using Bloom Filters to considerably reduce the memory required and an hysteresis mechanism to improve *steadiness*. Most strikingly this is achieved without impacting the *slice variance* of existing state-of-the-art protocols. In fact, SLEAD achieves similar performance regarding *steadiness* and *slice variance* with a fraction of the resources of existing approaches as shown in Figure 3.

The adaptation to churn in all the protocols studied in this paper is a direct consequence of the mechanism used to forget old node identifiers. Consequently, the removal of old identifiers is directly influenced by the frequency of view updates coming from the PSS and from the limited number of entries nodes are allowed to keep in memory. Surprisingly, both factors are not necessarily related to actual churn on the system, which hinders the capability of existing systems to perform well under heavy churn environments. We thus believe that a node removal mechanism that can take as a parameter the observed churn rate is essential to widen the range of applicability of distributed slicing protocols. This is an open problem, which we are trying to address using more complex Bloom

Filters [4]. The churn rate in a distributed large-scale system can be obtained through simple gossip-based mechanisms such as ChurnDetect [14].

References

1. P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 2007.
2. R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *International Workshop on Peer-to-Peer Systems*, 2003.
3. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.
4. K. Cheng, L. Xiang, and M. Iwaihara. Time-decaying Bloom Filters for data streams with skewed distributions. *International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*, 2005.
5. P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 2003.
6. A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal. Distributed Slicing in Dynamic Systems. In *International Conference on Distributed Computing Systems*, 2007.
7. J. Gantz. The Diverse and Exploding Digital Universe. Technical report, IDC White Paper - sponsored by EMC, 2008.
8. V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse. Sliver, A fast distributed slicing algorithm. In *ACM symposium on Principles of distributed computing*, 2008.
9. V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse. Slicing Distributed Systems. *IEEE Transactions on Computers*, 2009.
10. M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 2007.
11. M. Matos, R. Vilaca, J. Pereira, and R. Oliveira. An epidemic approach to dependable key-value substrates. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, 2011.
12. A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In *International Conference on Peer-to-Peer*, 2009.
13. A. Montresor, M. Jelasity, and O. Babaoglu. *Decentralized Ranking in Large-Scale Overlay Networks.* 2008.
14. Andrei Pruteanu, Venkat Iyer, and Stefan Dulman. Churndetect: a gossip-based churn estimator for large-scale dynamic networks. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 289–301, Berlin, Heidelberg, 2011. Springer-Verlag.
15. E. Rivière and S. Voulgaris. *Gossip-Based Networking for Internet-Scale Distributed Systems* . Lecture Notes in Business Information Processing. 2011.
16. H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 2005.
17. S. Voulgaris, D. Gavidia, and M. Van Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 2005.
18. F. Wang, Y. Xiong, and J. Liu. mTreebone: A Collaborative Tree-Mesh Overlay Network for Multicast Video Streaming. *IEEE Transactions on Parallel and Distributed Systems*, 2010.
19. M. Yoon. Aging Bloom Filter with Two Active Buffers for Dynamic Sets. *Ieee Transactions on Knowledge and Data Engineering*, 2010.