Slicing as a Distributed Systems Primitive

Francisco Maia, Miguel Matos and Rui Oliveira
High-Assurance Software Laboratory
INESC TEC & University of Minho
Portugal
Email: {fmaia,miguelmatos,rco}@di.uminho.pt

Etienne Rivière
Université de Neuchâtel
Switzerland
Email: etienne.riviere@unine.ch

Abstract

Large-scale distributed systems appear as the major infrastructures for supporting planet-scale services. These systems call for appropriate management mechanisms and protocols.

Slicing is an example of an autonomous, fully decentralized protocol suitable for large-scale environments. It aims at organizing the system into groups of nodes, called slices, according to an application-specific criteria where the size of each slice is relative to the size of the full system. This allows assigning a certain fraction of nodes to different task, according to their capabilities.

Although useful, current slicing techniques lack some features of considerable practical importance. This paper proposes a slicing protocol, that builds on existing solutions, and addresses some of their frailties. We present novel solutions to deal with non-uniform slices and to perform online and dynamic slices schema reconfiguration. Moreover, we describe how to provision a slice-local Peer Sampling Service for upper protocol layers and how to enhance slicing protocols with the capability of slicing over more than one attribute.

Slicing is presented as a complete, dependable and integrated distributed systems primitive for large-scale systems.

1 Introduction

The shift to large-scale distributed systems techniques is one of the main trends in the current computing era. This is linked with the advent of the Cloud Computing paradigm, which makes computational resources available at a global and ever-increasing scale. These systems call for the design of appropriate scalable and reliable distributed algorithms and protocols. However, designing such protocols and applications aimed at large-scale systems is a non-trivial task. The increase in system size is accompanied by escalated probability of software and hardware failures which need to be tolerated by the protocols.

In addition, system scale turns any kind of global knowledge assumption unrealistic. In fact, any mechanism that relies on information that grows linearly with the system size is unusable. Therefore large-scale systems require protocols designed to intrinsically scale to very large number of participating nodes, which should be able to cope with highly dynamic environments. These requirements are easily addressed by a well studied class of protocols known as *epidemic* or *gossip-based*. These have been used previously to build several Internet-scale systems and applications [23] like overlay construction and maintenance [9,27], consensus [20], data aggregation [17] and distributed slicing [8, 10, 21]. More recently, epidemic protocols have also been used in industrial systems such as Amazon's Dynamo [6] and Facebook's Cassandra [18].

There remain, however, several practical considerations that need to be taken into account to foster a broader adoption of gossip-based systems for large-scale systems management and operation. Such disregard of practical aspects often stem from the use of simplifying models and simulations. This gap was observed previously even in fundamental primitives such as consensus [4, 20]. We consider the same happens currently in another useful distributed systems primitive: *slicing*.

Slicing is the process of organizing a group of nodes into logical disjoint subgroups, called *slices*, according to some application dependent sortable criteria. Such logical division can be used for a variety of purposes such as the construction of hierarchical systems, identification of outliers, load-balancing or offering differentiated service levels [10]. Moreover, slicing is the natural candidate for managing heterogeneity which appears naturally in any large scale system from nodes with varying degrees of stability [1] to different resource capacities [13, 25]. As a matter of fact, popular systems such as the Skype VoIP service explicitly split the system into super and normal peers with different roles [12], and state-of-the-art video streaming systems like

mTreeBone [28] offload most of the work to stabler nodes to improve streaming quality. We thus believe there is a need for a distributed slicing primitive able to offer a generic but efficient slicing system that can be used by system and application designers.

Contribution: Since early work on slicing [8], research focused on providing a convergence proof, making it robust to churn and non-uniform Peer Sampling Services [10] and improve its resource usage and steadiness [21]. Nonetheless, these proposals still ignore a set of crucial requirements for any real system: multi-attribute slicing, non-uniform slice sizes, online slice reconfigurations and the ability to propose slicing as a substrate for other protocols by the provision of random set of nodes from the same slice, effectively implementing a slice-local *Peer Sampling Service* (PSS) [16].

In this paper we start by distilling existing protocols into a common slicing framework. Then, we present novel solutions to the aforementioned issues: dealing with non-uniform slices, performing online and dynamic slices schema reconfiguration, and provision of a slice-local Peer Sampling Service. We also describe how to enhance slicing protocols with the capability of slicing over more than one attribute. We discuss the impact of these additions to the various protocols proposed in the literature.

Roadmap: The rest of this paper is organized as follows: Section 2 motivates the paper. In Section 3 we provide a common slicing framework. Such framework is used to study and compare the different slicing protocols available. Moreover, we propose a variant of our previous work on SLEAD [21] illustrating the modularity of the framework. In Section 4 we extend the slicing framework with the requirements to make it a slicing primitive. Finally, we conclude the paper in Section 5.

2 Motivation

The motivation behind this work lies on the plurality of applications of distributed systems slicing. In order to illustrate our claim we describe three examples of application of slicing techniques.

The first example is related with critical services. Let us consider a large scale infrastructure consisting of a pool of computational nodes, shared by a set of heterogenous applications and services. This scenario is becoming relatively common thanks to the Cloud Computing paradigm and its use by Internet scale applications. Under these assumptions, the problem of resource assignment arises. For instance, in order to maintain acceptable quality of service levels and satisfy service level agreements, critical services or applications should be assigned to more stable nodes. Slicing allows partitioning the system according to some attribute such as the uptime of the nodes. In fact, it was demonstrated in previous research that nodes with a high uptime are more likely to be stable for another period [1]. Slicing can thus help solving the problem of critical service resource assignment by favoring these nodes.

A second example is data replication. Let us consider a simple data store system where the total amount of data can not be stored in a single system node. In order to guarantee that no data is lost, it is necessary to partition and replicate those data partitions across different system nodes. The problem is that of assigning data partitions to system nodes, without requiring costly protocols. Once again, slicing provides a simple yet robust solution. Slicing the system into a number of groups equal to the number of data partitions ensures that a virtually constant percentage of system nodes replicates each data partition.

Finally, a third example considers disaster recovery. Looking at the pervasiveness of geographic location information, it is possible to consider a system where each node has access to its geographic location. Slicing the system according to such attribute allows for location-aware slices. Let us also consider that each system slice is responsible for a set of data or services. On one side, it is possible to have slices with nodes that are physically close, to minimize communication costs. On the other side, having slices with nodes that are geographically distant can allow for replication of data or services that allows for disaster recovery. Even if a datacenter crashes, the slicing mechanism ensures that each slice has nodes that are geographically distant, replicating the correspondent data and services.

These are some examples of scenarios where a slicing primitive is an important asset. They serve as motivation for the present paper where we study current slicing protocols, compare them and propose a number of features that expand the usefulness of slicing.

3 Slicing framework and protocols

Autonomous and fully decentralized slicing using gossip-based protocols received some attention recently [8, 10, 21] due to its convenience and desirable properties for large-scale distributed system provisioning: dependability, scalability and adaptivity. However, little effort has been made to consider slicing as a building block for other applications and in particular, concerning its completeness. This capability is the key for composing gossip-based protocols into more complex services [22]. We begin by extracting from the existing literature the main characteristics of slicing protocols and factoring them in a generic slicing framework. Next, we go through existing slicing protocols, instantiating our framework to compare and differentiate them. Finally, we propose a variant of the SLEAD protocol [21] recurring to the modularity of the framework.

3.1 Basic Definitions

Slicing is the process of organizing the set of nodes in a distributed system, into k groups called $slices$. Each slice must eventually be composed of the nodes that lie in the sequence of k subgroups ranked by increasing values of a sortable metric: if slices are $S_1, \ldots, S_i, \ldots, S_k$, then all nodes belonging to S_i must have a greater value for the metric than those in S_{i-1}, and a lower value for the metric than the nodes in S_{i+1}. Examples of sortable metrics include the uptime, available disk space, CPU or other application-specific metrics. The system is modeled as a set of nodes connected through an asynchronous network.

Slicing protocols operate by means of gossip-based message exchanges of partial information about the system state, that yield a global convergence but do not require any centralized knowledge. In detail, each node in the system has access to a local attribute value representing the measured value of the metric of interest (disk space, uptime, etc.). Periodically, it contacts some peers and exchanges its attribute with them. Through this mechanism, each node gathers some local knowledge that it uses to progress. This is a key characteristic of gossip-based protocols that confers them high resilience and scalability.

The set of peers each node may contact (the *view* of each node) is given by an underlying protocol called the Peer Sampling Service (PSS) [16]. The PSS is typically implemented using gossip-based protocols itself and produces a random stream of peers drawn from the whole system. The PSS is a key service that maintains membership of nodes to the system in a decentralized fashion and offers a set of desirable properties, namely: i) departed nodes are eventually removed from the random stream of nodes provided at alive nodes, ii) new nodes are inserted in these streams within a bounded amount of time, and iii) convergence is guaranteed for protocols built on top of the PSS by ensuring that all nodes will be involved in the exchanges regularly. In our experiments, we assume the availability of the Cyclon [27] PSS implementation, that provides good randomness properties for the constructed views.

3.2 Slicing Framework

At this point, we can define the basic framework of a slicing protocol, with which we can instantiate the various existing slicing protocols. The pseudo-code for this framework is presented in Algorithm 1. In this algorithm, $node$ represents the node *id* while v represents its attribute value.

At its core are two threads, a passive (lines 2 to 6) and an active one (lines 8 to 10), running at each node. The active thread periodically and proactively sends to each neighbor in the $view$ a message containing the unique node identifier, me, and the current value of the local attribute, $local\_attribute\_value$ (line 10). Recall that $view$ is populated by the underlying PSS and is composed of a random subset of system peers (neighbors). The reception of those messages triggers the passive thread waiting condition. Upon reception, the slicing protocol stores the received information in a data structure (line 3) that offers three methods. The first method is $insertData(sender, attribute\_value)$ used to store incoming data. Methods $getSmaller()$ and $getTotal()$ refer to the attribute values the node has seen. The first one returns the number of attribute values which are smaller than the local one while $getTotal()$ returns the total number of attribute values received. Note that this only represents locally gathered information and does not requires global knowledge. With this local knowledge, nodes rely on an $estimate\_slice()$ (line 6) function to compute the node's slice and report it to the application.

What differentiates each instance of such framework is the possibility of implementing the $estimate\_slice()$ function and the data structure differently. Moreover, as we will see next, from the implementation details of both, the behavior and properties of each protocol change. However, in all protocols considered in this section the $estimate\_slice()$ function is implemented similarly (as shown in Algorithm 2). Existing literature on slicing only considers the case where every slice is equally-sized and each node in the system knows the number of slices, k, *a priori* by configuration. Consequently, computing the slice position is simply a matter of multiplying the node's position obtained from the ration of smaller attribute values and total nodes seen by k.

Algorithm 1: Slicing Framework.

```
1  function passive_thread
2      upon reception(message)
3          data.insert(message.sender, message.value)
4          smaller ← data.getSmaller()
5          total ← data.getTotal()
6          slice ← estimate_slice(smaller, total)
7  function active_thread
8      periodically
9          for (node, v) ∈ view do
10             send(sender = me, value = local_attribute_value)
```

Algorithm 2: Basic slice estimation.

```
1  function estimate_slice(smaller, nodes_seen)
2      position ← smaller/nodes_seen
3      slice ← position * number_of_slices
4      return slice
```

Algorithm 3: Data structures for RANKING.

```
1  initialization
2      smaller ← 0
3      total ← 0
4  function insertData(sender, attribute_value)
5      if ((attribute_value < local_attribute_value)
6          ∨(attribute_value = local_attribute_value
7              ∧sender < me)) then
8          smaller ← smaller + 1
9      total ← total + 1
10 function getSmaller()
11     return smaller
12 function getTotal()
13     return total
```

Algorithm 4: Data structure for SLIVER.

```
1  initialization
2      list ← new dict()
3  function insertData(sender, attribute_value)
4      list[sender] ← attribute_value
5  function getSmaller()
6      res ← 0
7      for k, v ∈ list do
8          if (v < local_attribute_value)
9              ∨(v == local_attribute_value
10                 ∧ k < me) then
11             res ← res + 1
12     return res
13 function getTotal()
14     return list.size()
```

To the best of our knowledge, there are three main slicing algorithms in the literature: RANKING [8], SLIVER [10] and SLEAD [21]. We describe each one of them instantiating the data structure implementation from our slicing framework and point out their specificities and motivations. Subsequently, in Section 3.6 we present a novel slicing protocol called DSLEAD.

3.3 RANKING

The RANKING protocol [8, 11] was the first slicing algorithm proposed in the literature. Its data structure simply consists on two variables: $smaller$ and $total$ updated each time a message is received. The pseudo-code for this data structure is presented in Algorithm 3.

It is important to note that the second part of the boolean expression in method $insertData$ (line 6) is used for disambiguation. Considering the possibility of two nodes sharing the same attribute value, their *id* is used to order the nodes, improving slice calculation.

Although very simple, the RANKING protocol is highly resilient. Message loss and churn do not prevent the protocol from progressing. However, some details prevent the protocol from achieving optimal results. In particular, there is no regard to duplicate messages. A node that receives duplicate messages from a specific peer will consider them repeatedly when calculating the slice estimative: the number of nodes with higher and lower values will thus be miscalculated leading to wrong slice attributions.

Observe that, in [8], a RANKING node contacts a single selected node at a time. However, RANKING can be implemented by sending the attribute to all nodes in the view. Such implementation is faster and avoids biasing the protocol towards nodes in the slice border [11]. In our framework, we only consider this version of the protocol.

3.4 SLIVER

To solve the duplicate message problem, the SLIVER [10] protocol was proposed. It is very similar to RANKING but alongside the node attributes, SLIVER also stores the node identifiers. This way, attributes from a specific node are considered only once in the slice estimation. The pseudo-code for SLIVER's structure follows on Algorithm 4. The data structure to support SLIVER is a key-value table where the keys are node ids and values their attributes (line 2).

It is possible to see that this protocol converges under the assumption of the availability of a PSS. Let us consider that a single node is capable of storing a number of pairs $(id, attribute\_value)$ equal to the size of the system. Due to the random nature and continuous refresh of the PSS views, eventually every node will receive a message from every other node in the system. With this global information available at each node it is easy to see that the protocol will converge and every node will compute the correct slice.

At this point it is important to make an observation. As noted in Section 1, a protocol that relies on an amount of information proportional to the system size is not scalable nor suitable to large-scale systems. To address this problem and to make SLIVER run in a bounded memory environment, instead of storing all received attribute values, only the more recent ones are kept. In practice, this is achieved by considering a FIFO queue with fixed size. It should be noted however, that this adjustment not only solves memory issues but also allows the protocol to handle churn in a more effective way. Nodes leaving the system will stop publishing their attribute values and the limited size queue will force them to be eventually forgotten from the system. Analogous behavior happens for nodes joining the system.

An alternative implementation of the RANKING protocol can also be considered in order to allow the protocol to forget attribute values. Instead of simply storing two variables, a fixed size list of attribute values is stored. The core behavior of the protocol is preserved but now it is able to cope with dynamic attribute values.

3.5 SLEAD

The SLEAD [21] protocol was proposed with the objective of tackling the lack of steadiness and high memory consumption issues manifested by previous approaches.

Steadiness issues were addressed using an hysteresis mechanism. In practice, the mechanism delays slice change decisions until such decisions have been confirmed by more than one cycle of slice estimation. This mechanism avoids unnecessary slice changes, specially for nodes at slice borders. The hysteresis mechanism, detailed in [21], is pluggable to all slicing protocols and is left out of the scope of the present paper.

Memory consumption problems arise from the need to store a list with every pair of $(id, attribute\_value)$ received in order to ensure that the protocol converges. Let us consider a stable environment where each node has a constant attribute value and no node leaves or enters the system. It is easy to see that, in such scenario, in order for a slicing protocol to converge to the correct slice organization it is necessary that each node *sees* the attribute value of every other node in the system. With that complete view over the system it is possible to compute the exact slice to which the node belongs. Nevertheless, having a protocol that uses an amount of memory proportional to system size is clearly not scalable. This problem was addressed with the use of a FIFO queue of fixed size, m, that follows the behavior of a sliding-window. As a result, at each point in time, every node has access to a sample of m pairs $(id, attribute\_value)$ with which it is able to estimate its relative position and thus its slice. Because the underlying Peer Sampling Service provides a stream of nodes that is extremely close to a continuous random selection, it is expected that each network sample preserves characteristics similar to the system as a whole distribution, hence allowing each node to correctly estimate its slice. However, in practice the size of m impacts the accuracy and steadiness of the protocol. Low values of m degrade the quality of the sample and negatively impact the protocols behavior while high values of m result in high memory consumption rates. This behavior is observable in the results from [21].

SLEAD's solution to the high memory consumption was the use of Bloom filters [2] to store data. With Bloom filters, SLEAD is able to store the complete view of the system with a bounded memory footprint which solves the problem for the case of a stable environment. Still, as noted in the SLEAD paper, the sliding-window-type behavior of SLIVER not only addresses memory consumption issues but also addresses system dynamism. The system may experience instability due to two main factors: churn or node-level change of attribute values. Both these factors provoke the need for the system to adapt and consider new information arriving and *forget* obsolete one. In SLIVER, this is immediately achieved through the fixed sized queue as old values are progressively being forgotten and replaced by fresh data. In SLEAD, as traditional Bloom filters do not have the capacity to remove items, a special kind of Bloom filters, called A^2 [29] is used. This Bloom filter implementation is capable of forgetting values by having two Bloom filters and periodically reseting one of them. Beside, we replaced the traditional Bloom filters used in A^2 by counting Bloom filters that allow for the removal of specific items [7]. The combination of both mechanisms enables SLEAD to cope with dynamism.

Algorithm 5: Data structures for SLEAD.

```
1  initialization
2      smaller ← new A2BloomFilter()
3      greater ← new A2BloomFilter()
4  function insertData(sender, attribute_value)
5      if ((attribute_value < local_attribute_value)
6          ∨(attribute_value = local_attribute_value
7              ∧sender < me)) then
8          smaller.insert(sender)
9          greater.remove(sender)
10     else
11         smaller.remove(sender)
12         greater.insert(sender)
13 function getSmaller()
14     return smaller.size()
15 function getTotal()
16     return smaller.size() + greater.size()
```

The instantiation of SLEAD in our slicing framework is presented in Algorithm 5.

3.6 DSLEAD: Decaying SLEAD

Although successful in reducing steadiness and memory consumption issues from previous approaches, the SLEAD protocol exhibits some frailties. In particular, the way it deals with dynamism with the A^2 Bloom filter implementation has a main disadvantage. related to the difficulty of configuring the rate at which values are being forgotten in a tractable way. This is important since forgetting values too fast results in protocol output instability while forgetting them slowly results in very slow adaptation to change.

In SLIVER, changing this rate is easy because it suffices to change the queue size (a larger queue means slower adaptation to change). In SLEAD this is not a straightforward task. The rate at which the protocol forgets values is related to the fill ratio of the A^2 Bloom filter [29]. Inevitably, this means that changing the refresh rate is achieved by changing the Bloom filter size. This is definitely not practical as changing the Bloom filter size means reseting the whole Bloom filter, impacting negatively on the slice estimation.

Fortunately, we can take advantage of the slicing framework and SLEAD modularity. In fact, SLEAD is independent of the Bloom filter implementation. To solve these issues and achieve a more complete slicing protocol we propose the use of a different Bloom filter variant: time-decaying Bloom filters [5]. These Bloom filters not only allow direct item removal but also allow the user to define a function that removes according to a certain time-related function. In other words, it is possible to define at which rate items are forgotten from the Bloom filter. This leads to DSLEAD, a variant of SLEAD that shares the same structure from Algorithm 5 but with a different Bloom filter implementation.

In our implementation it works as follows. Each time an item is inserted into the Bloom filter a certain number of positions in the Bloom filter array are incremented according to a set of hash functions [2]. It is important to note that the implementation of time-decay Bloom filters relies on counting Bloom filters [7] which have more than one bit per array position, allowing to count various occurrences of the same item and enabling item removals. Then, periodically, each of the array positions is multiplied by a $fraction$ value ($[0, 1[$). If a certain position or group of positions in the array are not refreshed their value will eventually decay to a value close to 0. As the value never reaches 0 and, in order to actually *forget* items, when a certain value in a certain array position becomes smaller than a user defined threshold, it is considered to be 0.

A decay function is, therefore, composed by three variables: the period of decay, the fraction of decay per period and the minimum threshold. The ability to define a decay function over the Bloom filter values and easily change the rate at which it is operating, alongside the ability to immediately accommodate changes to attribute values completes the DSLEAD protocol.

We evaluate both SLEAD and DSLEAD in the following experiment. We measure protocol steadiness as the number of slice changes that occur in the system for a particular cycle [21]. In our experiment we let the protocols run for about 50 cycles and then triggered a configuration change increasing the memory footprint of SLEAD and decreasing the decay rate of DSLEAD. The results are depicted in Figure 1. Note that we intentionally configured SLEAD and DSLEAD with small memory and high decay rate respectively which issue a non desirable behavior from both protocols, observable until cycle 48. In particular, the steadiness of slice estimation is degraded as values are continuously being *forgotten* or *decayed*. Both protocols eventually converge, lowering the steadiness values. However, DSLEAD does not incur the same burst of slice changes as SLEAD. This is due to the fact that in order to reconfigure SLEAD it is necessary to reset the protocol's data structure while this is avoided in DSLEAD, resulting in a much smoother transition.

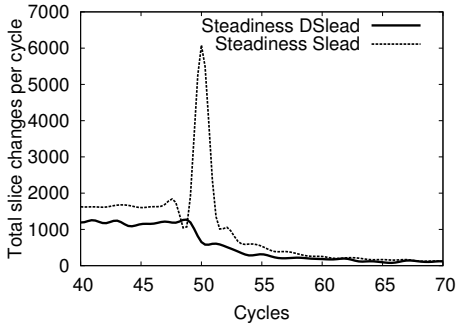

Figure 1: SLEAD and DSLEAD with reconfiguration.

4 Slicing as a Distributed Systems Primitive

Slicing protocols provide the capability of assigning each node a slice which is useful from the perspective of system organization. However, nothing is said about how nodes from the same slice interact with each other or how to have different sized slices. These are very important features for slicing protocols to be practical.

In particular, in our opinion, there are four main features that are lacking from slicing protocols. First, it is important to support slices with different sizes. This feature widens the range of applicability of slicing protocols. It considers, for instance, the case of having 10% of nodes with a coordination task and groups of 30% nodes each with separate responsibilities. Secondly, having a fixed slice configuration also impairs the applicability of slicing protocols. These protocols are intended for highly dynamic environments where adaptivity is key. Consequently, it is difficult to conceive the use of a protocol that needs to be preconfigured and restarted each time a new configuration is needed. We propose the addition of a mechanism that allows slicing protocols to change slice configuration without having to stop or restart the system. Third, in [15] it is noted that slicing is only useful if each slice is presented to the application as a group. This means that each node in the system should not only know to which slice it belongs to but also which nodes share such slice. Finally, the ability to slice considering more than one attribute seems a very useful feature. For instance, allowing to look not only to CPU

load as an indication of node utilization but to a combination of different metrics like disk I/O and uptime. In this section we describe each one of these features and how they can be implemented.

In addition, we evaluate some of these features. In this regard, we ran our experiments on top of Splay [19]. Splay is a platform that enables rapid development and testing of distributed systems. In particular, we ran each experiment on a local deployment of Splay and each consisting in 1000 Splay nodes. Splay was deployed in a 24-core machine with 128GB of RAM. Each node runs the same Lua [14] code consisting of the DSLEAD protocol. As described earlier, DSLEAD follows a gossip-like message exchange pattern. Each node periodically contacts its neighbors sharing its locally read attribute and considers this a *cycle*. However, nodes are not synchronized which means there is no guarantee that nodes are progressing, in terms of cycle count, at the same rate. In order to retrieve usable information from system runs these logical cycles are ignored and used only internally by each node. We present our results in terms of cycles measured in actual time. This period of observation was configured to be of 10 seconds and a cycle, in our experiments, should be understood as one of these periods.

4.1 Heterogeneous slicing

Previous work on distributed systems slicing protocols focused in dividing the system into k equally-sized slices. However, this approach is restrictive. Moreover, it is possible to equip existing protocols with the capability of considering different slice configurations with minimum change to the protocols and maintaining their properties. We name these slice configurations *schemas*.

Originally, each node calculates its slice by estimating its position in a virtual ranking of all nodes according to a certain attribute. This position estimative is calculated dividing the number of smaller attributes observed by the total number of attributes observed as described in the $estimate\_slice()$ function implementation of Algorithm 2.

To allow the protocol to consider different slice schemas, we need to store an additional data structure representing the slice size distribution: a simple schema configuration list (CL) with cumulative percentages is sufficient. This list is populated with one entry per slice (s_1 to s_k) and each entry, i, represents the percentage of the system expected to be assigned to all slices from s_1 to s_i. For instance, the list $CL \leftarrow [0.2, 0.4, 1]$, issues a system partitioned into three slices. The first slice would gather 20% of the system, the second slice another 20% of the system and the third slice would encompass the remaining 60% of system nodes. It is important to note that this organization still follows the virtual ranking of nodes according to a local attribute.

Furthermore, even though we are using DSLEAD in our experiments, this technique can be implemented by all the slicing protocols that fit the slicing framework we defined in Section 3 simply by reimplementing $estimate\_position$ as follows (Algorithm 6). The new function will still take as arguments the total number of attributes seen by the node and the number of those that are smaller than its local attribute. It computes the node's relative position in the virtual rank of all nodes as before but uses this result in a different way. The node slice is estimated by checking in which schema configuration interval such position falls.

Algorithm 6: Implementation of heterogeneous slice estimation.

```
1 initialization
2     CL ← list with slice configuration
3 function estimate_slice(smaller, total)
4     position ← smaller/total
5     slice ← 0
6     for s in CL do
7         if position <= s then
8             return slice
9         slice ← slice + 1
```

This change in slicing protocols proved to be effective. To evaluate this particular feature we ran two different tests with different slice schemas. Schema one considers five slices, each with 20% of the system nodes and it was chosen in order to show that previous equally sized slices are still achievable in this new protocol version. On the other hand, schema two is an heterogeneous slice schema. It is configured to achieve three slices, one with 50% of the nodes and the remaining two with 25% of nodes each. The results are depicted in Figure 2 and Figure 3, respectively.

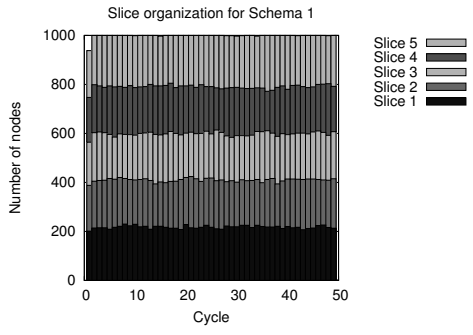

Figure 2: DSLEAD run configured with schema one.

Each vertical bar represents how the slices are distributed in a certain cycle. Each shade of grey represents the amount of nodes of a certain slice. Slices as ordered from bottom up according to the CL list order. The sum of all segments of

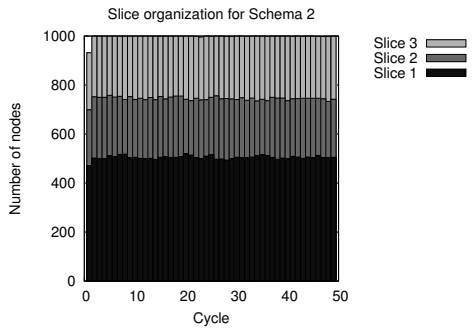

Figure 3: DSLEAD run configured with schema two.

the vertical bar is the total number of nodes in the system. Note that this is not true for some initial cycles, as can be observed in Figure 2 and Figure 3 where some nodes are still starting.

4.2 Slice reconfiguration

In previous work [8, 10, 15, 21] focused on slicing, the number of slices was a pre-configured parameter and little is said about dynamic reconfiguration of slices. Moreover, now that it is possible to have different slice schemas, the logical step is to capacitate slicing protocols with the ability to change schema *on-the-fly*.

We begin by assuming that the initiative of changing schema is external to the system and communicated to an arbitrary node or set of nodes. The nodes that receive the schema change request are responsible for processing it. The main challenge here is how to effectively disseminate the schema to all system nodes. However, all the slicing protocols rely on Cyclon [27]. Cyclon provides each node a random view over the complete set of nodes and this view has the properties of a connected graph. As a result, in order to send a message to all nodes in the system, it is sufficient to send it to all the nodes in the current Cyclon view and have each node repeat such task on the reception of a new message, essentially flooding the network. It is important to note that more elaborate and effective dissemination techniques could be used to spread the slice reconfiguration message [3]. Nonetheless, such messages are very small in size and sent sparingly and thus we consider such optimizations out of the scope of this work. The code for schema change request is presented in Algorithm 7.

In order to validate this approach we ran DSLEAD with an initial slice schema with $CL \leftarrow [0.1, 0.5, 0.6, 0.7, 0.9, 1]$ and issued a schema change to $CL \leftarrow [0.1, 0.2, 0.3, 0.4, 0.5, 1]$ at cycle 250. The change request was made to a single node in the system, responsible for propagating it.

To better illustrate the protocol behavior, Figure 4 depicts slice distribution at each ten cycle period and Figure 5 depicts slice variance measurements. Slice variance, as defined in [21], measures the distance of a certain slice distribution to a target distribution. In our experiment we determined the system slice variance according to both schemas. It is interesting to see how the two curves behave when the schema change occurs. Initially, as expected, high variance values are measured for schema two and low variance values measured for schema one. Around cycle 250, the curves exchange roles indicating the schema change. The same time behaviour can be observed in Figure 4. For instance, if we look at slice two, which held 40% of the nodes initially we can see how it shrinks to the 10% of nodes configured in schema two.

Algorithm 7: Implementation of $changeSchema$ function.

```
1 function changeSchema(newSchema)
2     if not CL == newSchema then
3         CL ← newSchema
4         for peerinview do
5             send(peer, newSchema)
```

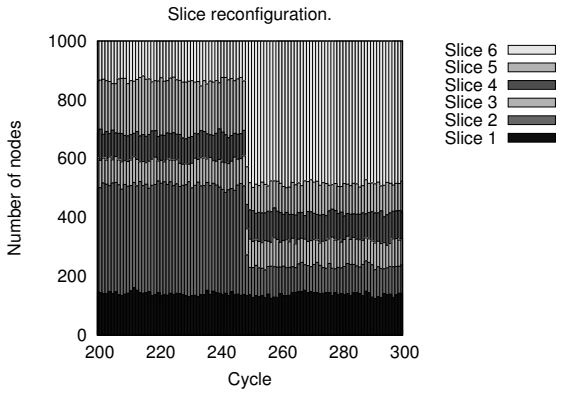

Figure 4: Slice reconfiguration.

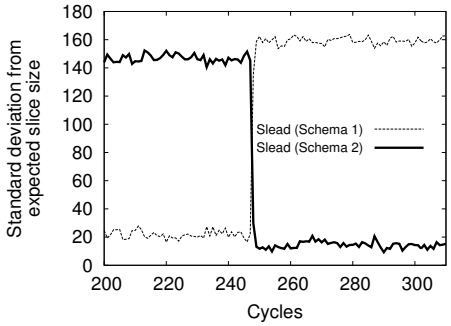

Figure 5: Slice variance for a slice schema change.

4.3 Slice-local view

Independent of the slice schema discussion, another gap in current slicing protocols is intra-slice connectivity. In the current slicing protocols, each node is capable of communicating with a set of other nodes in its $view$ which are called neighbors. This group of neighbors, although of key importance, is oblivious to the slicing protocol. Consequently, there is no practical way for a node to contact its slice peers. In fact, partitioning a distributed system into slices is only useful if it is possible to take advantage of such slices.

In order to achieve this the immediate solution is to have another instance of a Peer Sampling Service for each slice. This approach was actually mentioned in [15] as a future research path. This has however a bootstrapping problem as we do not know in advance each node's slice. Our approach is to inject the node's current slice into the slicing protocol message defined in Algorithm 1, line 10. Besides the current node's slice we also inject the slice-local view when the target node belongs to the same slice as the sender. This information, despite limited, allows nodes to quickly populate its slice-local view and quickly deal with changes by resetting it when slice changes happen.

4.4 Multi-attribute slicing

Existing slicing strategies take into account a single system attribute to rank nodes. However, it might be useful to consider more than one attribute in order to better characterize each node. For instance, considering various load attributes simultaneously or considering attributes such as geographic information and bandwidth simultaneously. The immediate approach can be to extend the slicing protocol to exchange more than one attribute at each algorithm cycle. This would allow considering various attributes for slicing. We propose instead a local computation of an artificial attribute resulting from the combination of different measured attributes and in particular the use of Space Filling Curves (SFC) [24]. This mathematical construction provides a mapping from a d-dimensional space to a unidimensional one. The different attributes considered are viewed as a multi-dimensional space. This space is divided into subspaces which are mapped to a line that traverses the subspaces passing through every point and entering and exiting the space only once. Then, the virtual attribute can be constructed as the length of the line from one of its ends to the point with spatial coordinates derived from the attribute values. Therefore, nodes will be able to map several attributes to a single point in the SFC given by a real number. This number is then used as the ranking criteria, allowing existing protocols to run unmodified but still supporting multi-attribute slicing. An example of the application of these curves can be found in [26].

5 Discussion

In this paper we focused on distributed systems slicing. Even though slicing has received attention from several researchers, the generalized use of these slicing techniques is impaired by the lack of some fundamental features. We look at slicing as a potential distributed systems *primitive*, i.e., as a potential building block for many applications.

To achieve our goal we start from the analysis of existing literature and define a generic slicing framework. The outcome of this study is twofold. On one hand, it allows a straightforward and exhaustive comparison of each protocol while, on the other hand, it lays down the foundation for future reasoning on these topics. In fact, our framework identifies the core features of a slicing protocol and identifies which are the components specific to each instantiation. As a result, in order to design and implement a new slicing protocol, it suffices to instantiate the slicing framework, which is done by implementing two components: the protocol's data structure and the slice calculation function.

Finally, we looked at slicing from a practical point of view and studied which features are missing towards the goal of having slicing as a distributed systems primitive. The first feature is non-equal sized slicing. Previously, slicing protocols divided the system into k equally sized slices. Because this is restrictive, we show how to implement heterogenous slicing. Secondly, we focus on dynamic slice schema change. Once it is possible to have heterogeneous slicing, it naturally follows the need for slice schema change without having to stop or restart the system. Finally, we equip slicing with a way to provide slices as groups and support to multi-attribute slicing.

An important point to make is that, although slicing as a building block is well defined, much work is yet to be done with respect to its application. For instance, the decaying mechanism introduced in DSLEAD can open very interesting research paths. In particular, properly defining decay variables is very important. These variables define the rate at which information is being forgotten from the system and replaced by fresh one. If we define the decay rate to be too high the system becomes very unstable while low decay values may imply slow adaptation to change. As churn is one of the main causes for dynamism, it could be interesting to find a correlation between the churn and decay rates. Such task in a non trivial one because churn rates are typically defined in terms of number of nodes joining/leaving the system while, due to how Bloom filters are implemented, knowing how many values are forgotten at each decay interval is hard. Nevertheless, finding a way to automatically adjust the slicing protocol according to the churn rate is a interesting and challenging problem.

Acknowledgements

The authors would like to thank Romain Rouvoy for our fruitful discussions which helped to improve the paper. This work is part-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 Stratus/FCOMP-01-0124-FEDER-015020; and FCT grants SFRH/BD/62380/2009 and SFRH/BD/71476/2010.

References

[1] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *International Workshop on Peer-to-Peer Systems*, 2003.

[2] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 1970.

[3] N. A. Carvalho, J. Pereira, R. Oliveira, and L. Rodrigues. Emergent Structure in Unstructured Epidemic Multicast. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2007.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live. In *ACM symposium on Principles of distributed computing*, 2007.

[5] K. Cheng, L. Xiang, and M. Iwaihara. Time-decaying Bloom Filters for data streams with skewed distributions. *Research Issues in Data Engineering: Stream Data Mining and Applications*, 2005.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS symposium on Operating systems principles*, 2007.

[7] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE-ACM Transactions on Networking*, 2000.

[8] 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.

[9] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Scamp: Peer-to-peer lightweight membership service for large-scale group communication. In *International COST264 Workshop on Networked Group Communication*, 2001.

[10] 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.

[11] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse. Slicing distributed systems. *IEEE Transactions on Computers*, 2009.

[12] S. Guha, N. Daswani, and R. Jain. An experimental study of the skype peer-to-peer voip system. In *International Workshop on Peer-to-Peer Systems*, 2006.

[13] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross. A measurement study of a large-scale p2p iptv system. *IEEE Transactions on Multimedia*, 2007.

[14] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of Lua. In *ACM SIGPLAN conference on History of programming languages*, 2007.

[15] M. Jelasity and A.-M. Kermarrec. Ordered slicing of very large-scale overlay networks. In *IEEE International Conference on Peer-to-Peer Computing*, 2006.

[16] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 2007.

[17] P. Jesus, C. Baquero, and P. S. Almeida. Fault-Tolerant Aggregation for Dynamic Networks. In *IEEE Symposium on Reliable Distributed Systems*, 2010.

[18] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[19] L. Leonini, E. Rivière, and P. Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Symposium on Networked Systems Design and Implementation*, 2009.

[20] F. Maia, M. Matos, J. Pereira, and R. Oliveira. Worldwide consensus. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2011.

[21] F. Maia, M. Matos, E. Rivière, and R. Oliveira. Slead: low-memory steady distributed systems slicing. In *IFIP International Conference on Distributed Applications and Interoperable Systems*, 2012.

[22] E. Rivière, R. Baldoni, H. Li, and J. Pereira. Compositional gossip: a conceptual architecture for designing gossip-based applications. *ACM SIGOPS Operating Systems Review, special issue on Gossip-based Networking*, 2007.

[23] E. Rivière and S. Voulgaris. *Gossip-Based Networking for Internet-Scale Distributed Systems*. Lecture Notes in Business Information Processing. 2011.

[24] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.

[25] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, 2002.

[26] R. Vilaça, R. Oliveira, and J. Pereira. A correlation-aware data placement strategy for key-value stores. In *International Conference on Distributed Applications and Interoperable Systems*, 2011.

[27] S. Voulgaris, D. Gavidia, and M. V. Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 2005.

[28] 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.

[29] M. Yoon. Aging Bloom Filter with Two Active Buffers for Dynamic Sets. *IEEE Transactions on Knowledge and Data Engineering*, 2010.