DataFlasks: epidemic store for massive scale systems

Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira
High Assurance Software Laboratory
INESC TEC and UMinho
Braga, Portugal
Email: {fmaia,miguelmatos,rmv,jop,rco}@di.uminho.pt

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

***Abstract*—Very large scale distributed systems provide some of the most interesting research challenges while at the same time being increasingly required by nowadays applications. The escalation in the amount of connected devices and data being produced and exchanged, demands new data management systems. Although new data stores are continuously being proposed, they are not suitable for very large scale environments. The high levels of churn and constant dynamics found in very large scale systems demand robust, proactive and unstructured approaches to data management. In this paper we propose a novel data store solely based on epidemic (or gossip-based) protocols. It leverages the capacity of these protocols to provide data persistence guarantees even in highly dynamic, massive scale systems. We provide an open source prototype of the data store and correspondent evaluation.**

***Keywords*—Dependability; Epidemic Protocols; Distributed Systems; Large Scale Data Stores;**

I. Introduction

Some of the most interesting challenges of our time emerge from the need to deal with very large scale systems. It is no longer unpractical to consider Internet-scale systems with thousands or even millions of nodes. Nevertheless, taking advantage of these resources is not trivial and scalability concerns arise. In fact, traditional protocols and system designs often turn out to be inadequate for these scenarios. This observation is particularly true for data management systems.

In this paper we address the problem of storing data in very large scale environments. We present a key-value store, DataFlasks, designed for massive scale systems. These systems exhibit two key characteristics that motivate our design and render current approaches ineffective. Firstly, massive scale systems are characterized by very high instability. With the increase in system size, faults and churn become the rule instead of the exception [1]. The second characteristic is the impossibility of attaining any kind of global knowledge at a single node. Note that any system that relies on global knowledge, which grows linearly with system size, cannot scale.

Recently, we have witnessed a surge of proposals for new data management systems, which typically are key-value stores [2]–[5]. When compared to traditional relational databases, these offer a simplified interface and reduced set of features. An example is the lack of transactional support. Having said that, these new management systems proved to be suitable for a significant number of tasks [6] and, notably, they are able to scale to deployments of hundreds of nodes. However, they are not able to perform well beyond those numbers. In fact, most of the new data management systems are based on structured peer-to-peer protocols [2]–[5]. In particular, they use a distributed hash table to organize nodes and distribute data among them. These protocols assume a moderately stable system and node churn greatly impacts their performance [7]. The fact these data stores do not handle churn effectively, greatly impairs their applicability to very large scale scenarios.

Instead, in the design of DataFlasks we propose the use of unstructured peer-to-peer protocols [8]. We built DataFlasks as a completely decentralized peer-to-peer system where there is no distinction between nodes. Every node runs the same set of algorithms and there is no hierarchy, structure or coordination of any kind. In particular, we designed DataFlasks based on a stack of gossip or epidemic protocols which are known for their scalability and resilience under highly dynamic environments. They have been successfully used to build several webscale systems and applications [9] like overlay construction and maintenance [10], [11], consensus [12], data aggregation [13], distributed slicing [14]–[17] and live video streaming [18]. One key characteristic of these protocols is that they rely solely on partial data about the system. In other words, each node makes progress by looking only at locally available data without any kind of global knowledge. The result is a key-value storage system that is able to scale to several thousands of nodes while, at the same time, cope with very high levels of node churn.

Contributions: We make three main contributions. Firstly, we propose a novel gossip-based algorithm for scalable and decentralized organization of system nodes into groups. This algorithm is part of the core of the DataFlasks providing data distribution and replication. Secondly, we propose a new key-value data store built entirely over gossip-based protocols aimed at highly dynamic environments. Finally, we provide an open source prototype of DataFlasks and evaluation of both the prototype and the group construction algorithm.

Paper layout: The paper is organized as follows. We begin by describing the design of our key-value store in Section II. We present a novel gossip algorithm that is at the core of DataFlasks and sketch a proof of its correctness in Section III. In Section IV we discuss extensions to the group construction algorithm that render the store effective in real systems. In Section V, we describe our implementation and provide evaluation of our prototype in Section VI. Finally, related work is presented in Section VII and Section VIII concludes the paper.

II. System Design

Along this section we present the design of DataFlasks. We describe the goals that drove our design and give the intuition behind how it works. Implementation details of the various components are left for subsequent sections.

DataFlasks is a key-value data store with a commonly used *put* and *get* interface. The *put* operation receives a key, an object and a version of the object and writes it to the store. Each stored object may be retrieved through a *get* operation that receives the object key and desired version as input. Every pair *(key,version)* is considered unique by the data store. The main goal of DataFlasks is to guarantee data persistence and availability.

Evidently, various versions of each object are possible for a single key. However, two concurrent write operations for the same *(key,version)* pair may lead to a data store inconsistency. In our design we assume that such concurrency control is handled by the client application of the data store. In particular, DataFlasks is designed to serve as a very large scale persistent layer for applications such as the ones described in [8], [19]–[22]. Once that concurrency control is addressed externally, it remains to describe how DataFlasks guarantees data persistence and availability. Moreover, how it does so for a system with several thousands of nodes with high levels of node churn.

The basic mechanism behind DataFlasks is the following. Any node may receive requests for *put* and *get* operations. When a *get* is received, if the node holds the correspondent data, i.e. holds the value correspondent to the requested key-version pair, it replies to the client. In the case of a *put* operation, the node can locally decide to store the data or not. Note that a node is allowed to decide not to store an object because we assume that a single node is not able to hold all key-value pairs. In DataFlasks, nodes are organized in groups and each group is responsible for a subset of the data. Nodes decide to store or discard data according to the group they belong to. Furthermore, organizing nodes into groups not only enables data distribution but also data replication. In fact, the size of the group will determine the replication factor for the data it holds. When a node stores the data as it lies in the corresponding group, it also propagates the request to the other members of the group for replication. Whenever a node is not able to satisfy a request, such request is epidemically disseminated to the other nodes.

Figure 1 depicts the basic mechanism of DataFlasks as well as its architecture. Four key components are present: request handler, store, group construction and communication. The request handler, as the name implies, is the component that allows the node to receive requests and process them issuing instructions to the other components. Each node has a store component that is responsible for actually persisting data. It abstracts the medium to which data is persisted which may vary for convenience. As described earlier, our data store relies on a novel group construction algorithm which is abstracted in the group construction component. This component is responsible for maintaining and making available to other components, information about the group the node belongs to. Finally, the communication component serves as an abstraction for a set of services needed by the other components in order to be able to exchange messages with other nodes in the system. In particular, it abstracts the request dissemination protocol.

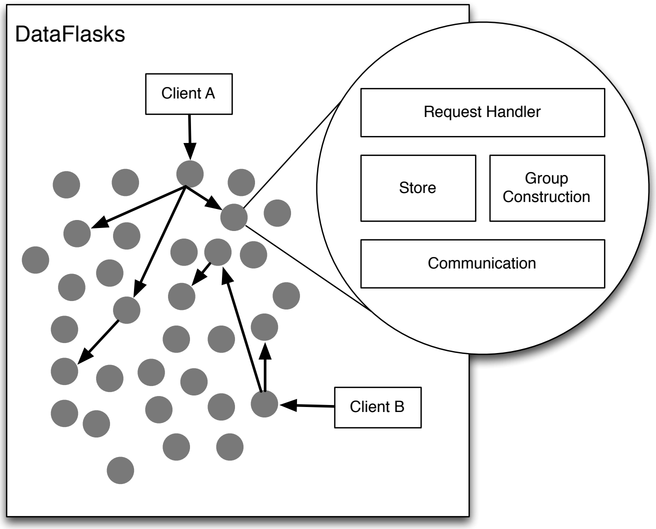


Fig. 1. Dataflasks Overview Architecture.

The two core challenges of DataFlasks are request dissemination and dividing nodes into groups. In our data store we rely on epidemic dissemination to route requests and on a novel gossip protocol for group construction. Epidemic dissemination has been the subject of extensive work and usable protocols exist [23]–[28]. On the contrary, the group construction protocol was designed specifically for DataFlasks and is the subject of Section III.

III. Group Construction

DataFlasks nodes must be divided into groups in order to distribute data and every node must learn to which group it belongs. In this Section we present an algorithm that is able to organize several thousands of nodes into groups in a robust and scalable way. The algorithm receives as an input the size of the groups to construct. This size is user defined and, by configuration, every node in the network learns the same desired group size at start up.

We begin with some remarks that give the intuition behind the design of the algorithm. Next, we present a simplified version of the algorithm and sketch a proof of its correctness. In subsequent sections we show how it can be extended in order to be faster and more effective.

A. Design

In the design of the protocol it is important to take into account that its main goal is to divide nodes into groups for data distribution and replication. In this scenario, each time a node changes group it needs to perform state transfer procedures. The design of our group construction algorithm aims at minimizing these procedures, which are costly.

Algorithm 1: Determining to which group a certain key-value pair belongs.

```
1 Method group(key):
2     $key\_hash \leftarrow hash(key)$
3     $key\_position \leftarrow key\_hash/hash\_max\_value$
4     $group \leftarrow \lceil key\_position * ngroups \rceil$
5     $return\ group$
```

At each node, the algorithm provides an estimation of the number of groups needed to satisfy the desired group size (or, in this context, replication factor) and, from those groups, the group the node belongs to. Data distribution is done by assigning key-ranges to each group. This is achieved by determining to which group a certain key belongs following the procedure of Algorithm 1. We assume the existence of an hash function ($hash$ in line 2). The hash function maps the keys with arbitrary range size into a fixed size range, trying to do so as evenly as possible over the target range. Assuming the target range is]0,hash_max_value], it is possible to map each key to a position in the range]0,1] (lines 2 and 3). With this mapping, it is straightforward to calculate the group a key belongs to (line 4). Note that, following this procedure data is distributed and balanced throughout the various groups and every node can locally determine if a certain key-value pair belongs in its data store.

Using this data to group mapping, we designed the algorithm to always consider the number of groups to be a power of two. Consider Figure 2. Forcing the number of groups ($ngroups$) to be a power of two, results in a well defined set of possible group configurations. Each configuration is associated with a level number where $ngroups = 2^{level}$. An important thing to notice is that the mapping between the key and group is stable as the level increases. Once each key is mapped to a]0,1] range, its position is preserved across different configuration levels as depicted by the black arrow in Figure 2. The goal of this design is to minimize state transfers between nodes every time there is an group change. In fact, when a configuration level is increased, nodes do not need to transfer any data. Deleting spurious data is even optional and may be performed only if space is needed. Conversely, when a level is decreased, state transfers are made only between pairs of groups distributing and balancing the task.

B. Algorithm

Typically, a gossip protocol works as follows. Every node knows a set of other nodes in the network, which we call *view*. Periodically, each node contacts one ore more nodes in its view and shares knowledge with them. Through these periodical exchanges each node is able to gather sufficient information to progress. Strikingly, many gossip protocols are effective even if the size of the view only grows logarithmically with the size of the system. This characteristic renders these protocols highly scalable. Even so, the node view must be populated. This problem is addressed by a specific class of protocols, which are themselves gossip protocols and which implement a Peer Sampling Service [10], [29]–[31]. These protocols provide each node with a random stream of peers which is used to populate the node view. Our group construction protocol assumes the existence of such a service. In particular, we consider Cyclon [10] as the Peer Sampling Service.

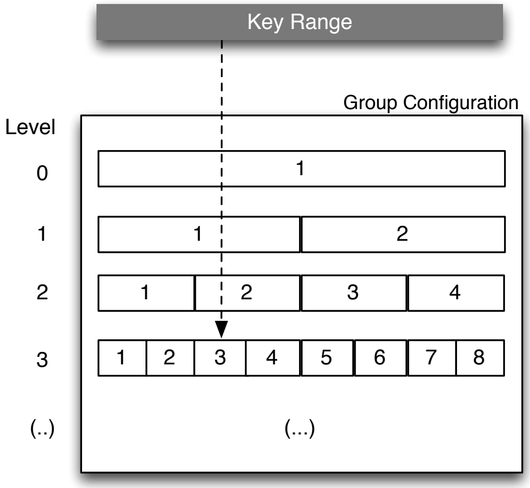


Fig. 2. Data to group mapping and group levels.

Cyclon works by periodically exchanging messages containing a set of random node references from the network. These references contain the information needed to contact the corresponding nodes. For the purpose of simplicity, we consider that these references are the node identification numbers and that knowing a node identification number (*id*) is sufficient to be able to contact it. In our protocol we leverage the existence of the PSS taking advantage of the messages it exchanges. Each time a PSS message containing a random set of peers from the network is received, a copy of this message is delivered to our algorithm. Our protocol reacts to the reception of such messages and, solely based on them, converges to the desired groups configuration. We consider that nodes are completely connected through lossy communication channels [32].

The simplified version of the group construction algorithm is presented in Algorithm 2.

The protocol has two parameters. The desired group size and the current node identification. Every node in the system runs the same protocol and is initialized with the same group size (*groupsize*). The node identification (*id*) uniquely identifies each node.

Upon initialization every node considers the system as a single group and that it belongs to that group. To this end it initializes variables *ngroups* and *group* with the value 1. The former variable stores the number of groups the node estimates should exist to comply with the desired group size. The latter stores the estimation of the group the node belongs to. Additionally, each node has a list variable (*localview*), initially empty, where it stores peers that belong in his group. As the protocol runs, the estimation of *ngroups* converges towards a number that divides the system into groups of *groupsize* nodes.

An important initialization step is generating the node position (*pos*). The node position is a number in the interval $]0, 1]$ generated at node start and that remains constant while the node is alive. This value allows nodes to distribute themselves into groups. The position is calculated using a number

Algorithm 2: Gossip group construction algorithm.

```
   input   : groupsize, id
   Data: $float\ pos \leftarrow random()$
   /* random number in $]0,1]$                              */
   Data: $ngroups \leftarrow 1$
   Data: $group \leftarrow 1$
   Data: $set\ localview \leftarrow \{\}$
1  upon reception of m ← set of (id, pos) from PSS:
       /* add new peers to $localview$                      */
2      foreach $peer\ in\ m$ do
3          if $group(peer.pos, ngroups) == group$ then
4              $localview = localview \cup \{peer\}$/* possibly
                   rewriting peer                             */
       /* clean $localview$                                  */
5      foreach $peer\ in\ localview$ do
6          if $group(peer.pos, ngroups)! = group$ then
7              $localview = localview \setminus \{peer\}$
       /* need to merge or split?                            */
8      if $|localview| < groupsize$ then
           /* Should Merge.                                  */
9          if $ngroups > 1$ then
10             $ngroups \leftarrow ngroups/2$
11     if $|localview| > groupsize$ then
           /* Should Split.                                  */
12         $ngroups \leftarrow ngroups * 2$
       /* recalculate my group                               */
13     $group \leftarrow group(pos, ngroups)$
```

Algorithm 3: Group calculation method.

```
1  Method group(position, ngroups):
2      $group \leftarrow \lceil position * ngroups \rceil$
3      return group
```

generator, which we assume is uniformly random across the entire network. Note that, with the node position and knowing the number of groups (*ngroups*) it is trivial to calculate the group to which the node belongs. The node position places the node in a range]0,1]. Consequently, to calculate the node group it suffices to divide such range into *ngroups* smaller ranges and determine in which of those the node position fits. Moreover, by the uniformity of the number generator, nodes will be evenly distributed across groups. The node group calculation is abstracted in line 13 of Algorithm 2 and shown in Algorithm 3.

DATAFLASKS group construction algorithm works as a passive thread that waits for messages from the Peer Sampling Service, which contain references for other nodes in the network. In DATAFLASKS, node references also include the node position. Recall that the node position is calculated only once and remains unchanged while the node is alive. It is thus safe to disseminate the position alongside the node *id*.

Upon the reception of a PSS message (line 1), the protocol performs four tasks. First, for each node reference in the message it checks if the correspondent node belongs to the same group (lines 2 to 4). If it does, it adds such reference to its local view (*localview*). Second, it checks if every reference in its local view still belongs to the same group (lines 5 to 7). This is necessary because nodes may change their estimation for *ngroups*. Consider a scenario where a node estimates that the correct value for *ngroups* is 2. In that case, half of the system nodes belong to the same group as the node. However, if the node refines its estimation to a value of *ngroups* of 4, then only a quarter of the system nodes can now belong to its local view. Following this process, *localview* holds references for peers that each node estimates to belong in its group. Consequently, the size of *localview* is the group size estimation at each node. At the third step of the algorithm (lines 8 to 12) such group size estimation is compared with the *groupsize* defined by the user. If the current size of $localview$ is smaller or greater than *groupsize* the node refines its estimation of *ngroups* in order to correct such violation. For the case it is greater that desired, *ngroups* is multiplied by two in order to lower the group size. We name this operation a *split*. Inversely, nodes perform a *merge* operation when there are insufficient nodes in the group. Finally, after adjusting *ngroups*, each node recalculates the group it belongs to (line 13).

With the continuous arrival of PSS messages the protocol continuously improves the estimation for *ngroups*. In the remainder of the Section, we present a proof of correctness for the simplified version of the protocol and present simulation results that show it converges to the correct group configuration.

C. Proof of correctness

The objective of Algorithm 2 is to group an arbitrary large number of nodes into sets of size $groupsize$ (being $groupsize$ the desired replication factor). In the following we sketch the proof that given a stable membership then the algorithm eventually converges and stabilizes.

Let us assume N nodes, such that $\frac{N}{groupsize} = 2^{level}$ for some $level \geq 0$. These nodes do not fail or leave the system. Nodes are fully connected by lossy communication channels, have access to a Peer Sampling Service that provides each node with a periodical random sample of nodes from the entire system, and also to a uniform random number generator in the interval]0,1].

Each node manages a variable $ngroups$. We show that, starting with $ngroups = 1$, each node i will eventually reach $ngroups = \frac{N}{groupsize}$ and stabilize there. We do so by firstly 1) showing that the algorithm has an upper bound $\frac{N}{groupsize}$ on the number of groups it can split the system into, then that, 2) at each node, $ngroups$ cannot be indefinitely smaller than $\frac{N}{groupsize}$, and finally that 3) eventually, once $ngroups = \frac{N}{groupsize}$, $ngroups$ no longer changes.

In the following, consider that for each level l, j is a neighbor of i if it belongs to the same group of i at l. From the group calculation $group \leftarrow \lceil position * ngroups \rceil$ (line 2 of Algorithm 3) if a node j is a neighbor of i at level l then it is a neighbor of i for every level k where $k < l$.

1) The algorithm has an upper bound $\frac{N}{groupsize}$ on the number of groups it can split the system into. Assume not, that is, eventually $ngroups > \frac{N}{groupsize}$.

Let $2^g = \frac{N}{groupsize}$. Once $ngroups > \frac{N}{groupsize}$ then the node is at least at level $g + 1$. It means that the node has performed a split at level g, which means that $|localview| >$

$groupsize$ at level g. However, this is not possible since for 2^g groups with N nodes there are at most $groupsize$ nodes per group. A contradiction.

2) At each node, $ngroups$ cannot be indefinitely smaller than $\frac{N}{groupsize}$. Again, for a contradiction, assume that $ngroups < \frac{N}{groupsize}$ is always true.

As $ngroups < \frac{N}{groupsize}$ then i must be at some level $k < g$. Because any neighbor of i at level k is also a neighbor of i at any level $j < k$, then by the PSS properties all neighbors of i at level k will be eventually addded to i's $localview$. These nodes will not be removed from i's $localview$ (lines 5 to 7) while i is at any level $j \leq k$. Since the neighbors of i at any level $k < g$ is larger that $groupsize$, i's $localview$ at level k will eventually grow larger than $groupsize$ and i splits. At level $g - 1$ i eventually splits and $ngroups = \frac{N}{groupsize}$. A contradiction.

3) Eventually, once $ngroups = \frac{N}{groupsize}$, $ngroups$ no longer changes.

Let $2^g = \frac{N}{groupsize}$. Because any neighbor of i at level g is also a neighbor of i at any level $k < g$, then by the PSS properties all neighbors of i at level g will be eventually added to is $localview$. These nodes will not be removed from i's $localview$ (lines 5 to 7) while i is at any level $k \leq g$.

Once, by 2) i reaches level g and all its neighbors at level g belong to $localview$ then i no longer merges (lines 8 to 10). And by 1), i never reaches any level larger than g. Therefore $ngroups$ no longer changes and the node stabilizes.

D. Convergence

In order to validate the convergence of our algorithm we ran a simulation[1]. In this simulation we considered 10.240 nodes and a Peer Sampling Service that delivered messages with random references of nodes. Additionally, uniformly distributed position values were generated for every node and *groupsize* defined to 10. For this simulation in particular the correct number of groups (*ngroups*) is 1024.

At each simulation cycle a single PSS message was delivered to each node to be processed. The size of the PSS message influences directly the speed of convergence of the protocol. Typically, the message size increases logarithmically with the system size [24]. We considered PSS messages containing 20, 30, 40, 50 or 100 node references[2]. In Figure 3 we depict the results of the simulations. The plot shows the percentage of nodes that hold a wrong estimation for *ngroups* per cycle.

From the results we can verify that the protocol converges to the desired configuration. It is also possible to see that, as expected, increasing the PSS message size improves the performance of the protocol. Nevertheless, note that it converges even for very small message sizes with respect to the size of the system.

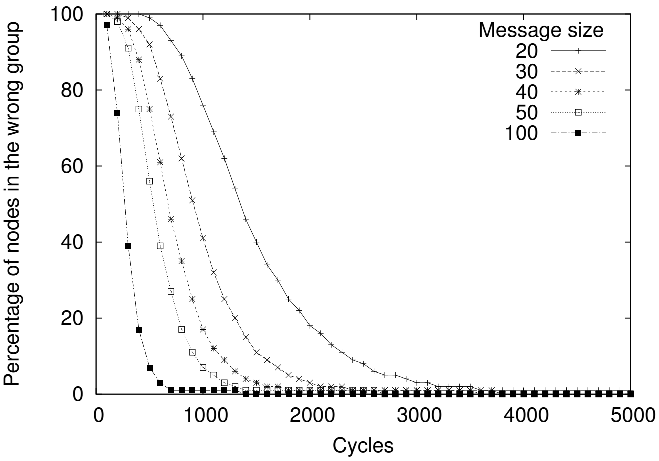


Fig. 3. Convergence of 10.240 nodes running the simplified version of the group construction algorithm.

E. Limitations

In this simplified version, the algorithm has also two important limitations. On one hand, if the desired group size does not exactly divide the number of nodes in the system in such way *ngroups* is a power of two, the algorithm does not stabilize. For instance, if in simulation of Figure 3 the number of nodes was 10.240 plus one the system would not stabilize completely. One group would detect an extra node in the system would not stabilize for any estimation of the number of groups. On the other hand, nothing is said about how it deals with churn. These two simplifications allow us to convey the main intuition behind the protocol in a straightforward way. Extensions to the basic protocol are proposed in Section IV to solve both limitations.

IV. Extensions

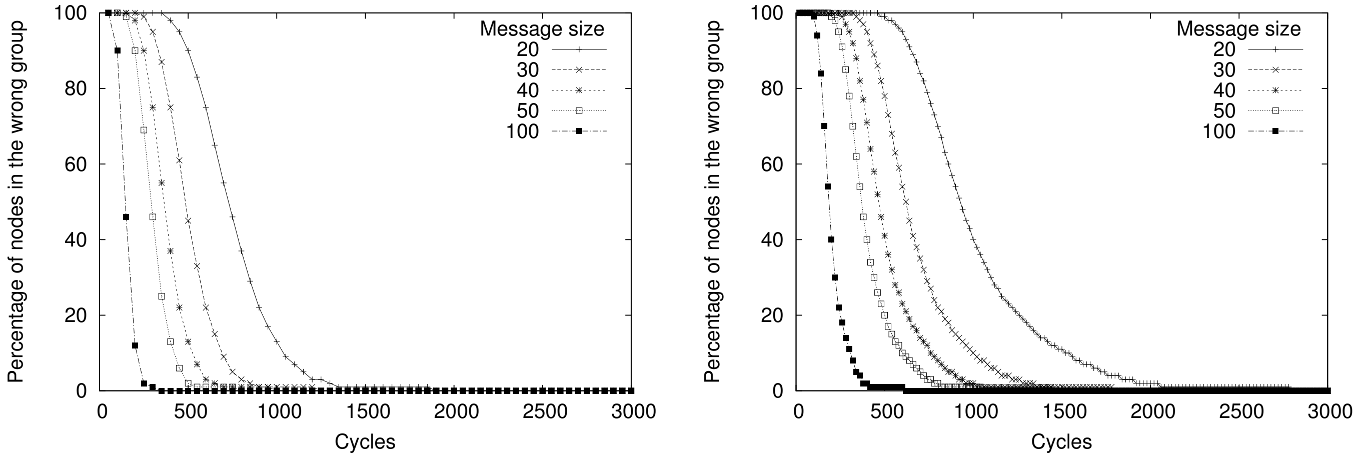
So far, the presented algorithm applies to a hypothetical system absolutely stable and with a round number of nodes. In practice however, the algorithm most probably never stabilizes but instead adapts to the dynamics of the membership. In this Section, we describe extensions to the algorithm of Section III in order to overcome the limitations identified previously. The first extension allows the protocol to support arbitrary system sizes. Next, we describe how the protocol can be extended to be able to handle churn.

A. Handling arbitrary system sizes

As highlighted previously, the simplified algorithm presented in Section III is very sensitive with respect to the defined group size. The algorithm stabilizes if and only if there exists a power of two, *ngroups*, that exactly divides the system size in groups of *groupsize* nodes. The fact is that aiming at an exact group size (lines 8 to 12 in Algorithm 2) is restrictive. Moreover, for the type of systems we are considering knowing the exact size of the system is unfeasible. In order to tackle this limitation, we extend the algorithm in order to allow the definition of minimum and maximum group size thresholds. This simple enhancement allows the protocol to converge in real case scenarios. Algorithm 4 depicts the changes needed to add this extension.

[1] Code used for simulations is available at github.com/fmaia/dataflasks_sim

[2] Note that the protocol converges even with smaller message sizes however, considering the system size, smaller messages lead to slow convergence.



(a) Simulation of the flexible group size mechanism with 10.240 nodes.

(b) Simulation of the flexible group size mechanism with 15.000 nodes.

Fig. 4. Group construction convergence extended with flexible group size.

Algorithm 4: Extended group construction algorithm.

```
input   : min_groupize, max_groupsize, id
(...)
upon reception of m ← set of (id, pos) from PSS:
    (...)
    /* need to merge or split?                    */
    if $|locaview| < min\_groupsize$ then
        /* Should Merge.                          */
        if $ngroups > 1$ then
            $ngroups \leftarrow ngroups/2$
    if $|localview| > max\_groupsize$ then
        /* Should Split.                          */
        $ngroups \leftarrow ngroups * 2$
    (...)
```

In order to validate that the proposed extension does not impair the convergence of the protocol we conducted two simulations. In the first simulation, the conditions were similar to those of Section III for a system with 10,240 nodes and configured $min\_groupsize = 5$ and $max\_groupsize = 15$. As can be observed in Figure 4(a) the protocol converges preserving the desired behavior of the simplified version. Strikingly, the algorithm converges even faster when configured with this extension. This is due to the fact that splitting and merging decisions are delayed due to the threshold flexibility. Such delay allows the node to preserve more node references in its local view per cycle. These references allow better splitting and merging decisions speeding up the convergence process.

We then ran a second simulation. In this simulation we used the same configurations of the previous one but, this time, for a system with 15,000 nodes. As observable in Figure 4(b), the protocol still converges to the desired configuration.

It is important to note that the maximum replication factor value must be at least double of the minimum. As the number of groups is always a power of two, choosing the replication thresholds this way avoids frontier cases where the system may enter a cycle of consecutive merge and split operations.

B. Handling churn

In order for the group construction algorithm to be useful it needs to be able to handle system dynamics. Nodes that leave the system must be eventually removed from every node's view while nodes entering the system must be incorporated. As stated previously, the group construction algorithm assumes the existence of a Peer Sampling Service. By design, the Peer Sampling Service is able to handle system dynamics. Nodes that leave the system are eventually removed from every sample the PSS delivers. Joining nodes, eventually, are sampled with the same probability as any other node in the system.

Having said this, the problem of handling nodes joining the system is immediately solved. New nodes are eventually sampled by the PSS and incorporated into the group construction algorithm. However, as it is, the group construction algorithm is unable to handle node departure. In fact, even if not subsequently sampled by the PSS, departure nodes for which there is a reference stored in a node view are never *forgotten* if they belong to the same group as the node. This limitation can be overcome introducing an aging mechanism for node references. Each node reference is now tagged with an age property. When a node reference is delivered by the PSS it is tagged with age 0. Each time a new PSS message is delivered, every node sees its age increased by 1. With this extension, it is now possible to define a maximum age threshold to allow node references to be forgotten when obsolete. Note that, if a node leaves the system eventually ceases to be sampled by the PSS. Consequently, every of its node references stored in any of the active nodes will inevitably age beyond the age threshold and eventually be forgotten. As a result, the protocol is now able to handle node departure.

Nevertheless, it is necessary to define an adequate maximum age threshold. Intuitively, such threshold must be higher than the number of cycles required to refresh a reference to a valid node. Note that, if this was not the case, valid references would be continuously removed impairing the convergence of the algorithm. In practice this means defining the age threshold superior to the time the PSS needs to sample a certain node reference. Unfortunately, this is too slow. As the system size grows, the probability of a certain node to be sampled by the PSS in each cycle decreases. Consequently, the number of cycles needed to make sure a certain node is sampled becomes unmanageably high.

In DataFlasks, we tackle this limitation by adding an active thread to the group construction algorithm. Periodically, each node produces a node reference to itself with age 0. It then sends such reference, alongside with all the references in the local view, to all the nodes it estimates to be in the same group. This simple mechanism, allows refreshing node references. Note that, once a node has left the system, every reference to it that may exist in the system will stop being refreshed. Eventually, it is removed from every node's local view as desirable. The active thread may be seen as an *heart beat* mechanism. An important thing to notice is that, although this mechanism is not required for convergence, it improves significantly the algorithm's speed of convergence. As nodes exchange references with nodes from the same group periodically, nodes receive useful references without waiting for the PSS to sample them all. Note that, as described in [33], the Peer Sampling Service randomness properties are essential but, typically, not sufficient to achieve good convergence results.

Figure 5, depicts the results for a simulation of the group construction algorithm with the all the extensions described so far. The simulation was configured with 15,000 nodes, $min\_groupsize = 5$, $max\_groupsize = 15$ and different view sizes. Additionally, the maximum age threshold was defined to 30 and the active thread is launched every 15 cycles. As observable, the algorithm still converges to the desired organization and is now much faster.

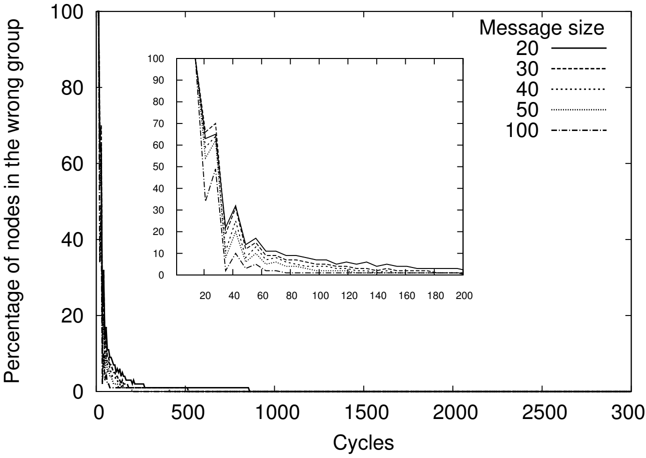


Fig. 5. Convergence of 15.000 nodes running the extended version of the group construction algorithm.

It remains to access the actual ability of the algorithm to reconfigure itself when there is a massive departure or entrance of nodes. Note that, with the extension considered for handling arbitrary system sizes, moderate system dynamics are inherently handled. In order to force system reconfiguration it is necessary that the number of nodes entering or leaving the system be sufficient to force a split or merge operation respectively. In Figure 6, we depict the results for two simulations. Figure 6(a) presents the results for an experiment where 7,500 nodes are added to a stable system of 7,500 nodes. Such membership change is made at cycle 500. The line in the Figure represents the percentage of nodes with a wrong estimation for the number of groups. Note that the expected estimation changes at cycle 500. In this case, the configuration dictates that initially the correct number of groups is 512. With the addition of another 7,500 nodes, the number of groups is expected to be 1,024. Consequently, until cycle 500, the line depicts the percentage of nodes that do not estimate the number of groups to be 512, after such cycle it depicts the percentage of nodes that do not estimate it to be 1,024. As it is observable, after the abrupt addition of nodes the algorithm is able to converge to the new configuration.

In an analogous experiment, 50% of the nodes were removed from a 15,000 node stabilized system. The results are depicted in Figure 6(b). In this case, until cycle 500 the plot depicts the percentage of nodes that have a number of groups estimation different from the 1,024. After cycle 500, it depicts the percentage of estimations that are not 512. In fact, the removal of 50% of the system nodes forces the algorithm to perform a merge operation in order to preserve the replication factor above the minimal threshold value. Similarly to the previous experiment, the algorithm is able to converge as desired.

V. Implementation

Having validated the core components of DataFlasks it remains to show it preserves data persistence. For this purpose we implemented a complete DataFlasks prototype in Java[3]. Additionally, we implemented a DataFlasks binding to the Yahoo! Cloud Serving Benchmark (YCSB) [34]. Along this Section we discuss some implementation details of the prototype and the YCSB binding.

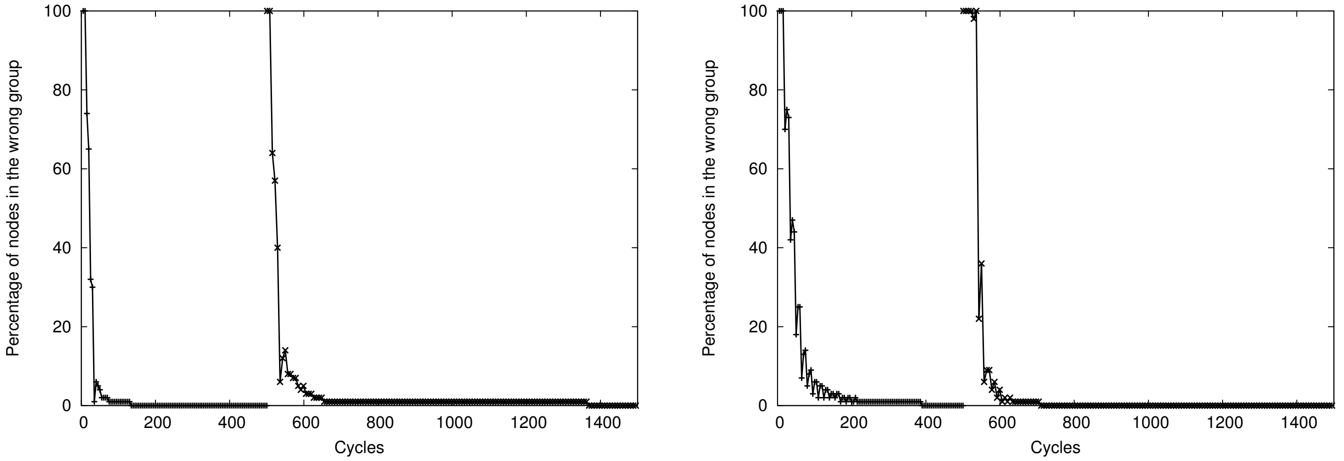
Our Java implementation roughly follows the architecture depicted in Figure 1. Nodes communicate through UDP sockets and the current prototype assumes all nodes to be in the same network. Such assumption allows the nodes to contact each other knowing only their IP addresses. Problems arising from removing this assumption are out of the scope of the present paper.

The group construction component includes the algorithm described in Section III with all the extensions of Section IV and a version of the Cyclon algorithm [10] serving as Peer Sampling Service. The prototype also includes a boot component that initiates the Cyclon protocol. Even though this is a simplification, it doesn't affect the behavior of DataFlasks . It replaces the node joining procedure described in [10] by providing initial random sets of peers to populate the node's initial view.

The store component is currently an in-memory store. Our prototype is modular and the store can be replaced with a disk based one, however, because in our evaluation we consider that when a node fails all data is lost, an in-memory store is adequate.

For data dissemination, the current implementation of DataFlasks uses a combination of two mechanisms. Firstly,

[3] The DataFlasks prototype is open source and available at github.com/fmaia/dataflasks



(a) Simulation of 7,500 nodes. Number of nodes doubled at cycle 500.

(b) Simulation of 15,000 nodes. At cycle 500, 7,500 nodes are removed from the simulation.

Fig. 6. Simulations of the group construction algorithm subject to churn.

each request is disseminated in an *infect and die* model [24]. To this end, each node forwards each request to the peers in their Cyclon view. The size of the view is considered to be large enough to provide good dissemination properties as described in [24]. Note that, requests are not required to reach all nodes in the system. It is sufficient to ensure it reaches some nodes of each group. Moreover, once a request reaches a node belonging to the group of interest, such node disseminates it to the nodes in its group. Secondly, to ensure data reaches all nodes that must replicate it, DATAFLASKS uses an anti-entropy mechanism [23]. This mechanism allows nodes to periodically restore possibly missing data replicas.

Finally, we note the following important aspect of the YCSB binding. The YCSB system must be able to discover DATAFLASKS nodes in order to issue data requests. This functionality is provided by a Load Balancer component in our prototype. Currently, this component provides YCSB with random nodes sampled from the system. This is sufficient for our proof of concept but many optimizations may be implemented in this component. The most simple one would be caching. Once a node has satisfied a request for a certain key, i.e. belongs to a certain group, it may be stored with such information. Subsequent requests can then be judiciously routed to appropriate nodes.

VI. EVALUATION

Having described our DATAFLASKS prototype, in this Section we present a set of experiments to validate persistence guarantees of our system. Our prototype is written in Java code and not binded to any specific simulation framework. However, we do not have sufficient machines at our disposal for the scale we target. Consequently, we used the *Minha* framework [35]. This framework emulates various Java Virtual Machines on top of a single one which makes it possible to run multiple hosts on a single machine. In our case we used a machine with an AMD Opteron 6172 (24 core at 2.1GHz) and 128GB of memory. As a result, we are able to evaluate DATAFLASKS in a very large scale environment while, at the same time, provide a Java prototype that is actually ready for deployment in a real scenario.

In our experiments we ran 1000 nodes and populated DATAFLASKS with 200.000 data objects. DATAFLASKS was run with the extended group construction algorithm. We configured the Peer Sampling Service to exchange messages every 2 seconds and the active group construction mechanism every 15 seconds. Additionally, the anti-entropy mechanism was configured to run at intervals of 30 seconds.

We subjected the system to different levels of churn and recorded the number of replicas per key each 10 seconds. Churn is implemented by removing a node and adding a fresh one preserving the position distribution of the nodes removed. We assume that churn is uniform across the entire system, which in a very large scale scenario means that nodes leaving and entering are uniformly distributed by all groups. Consequently, in our experiments, subjecting the system to 10% of churn means subjecting each group to 10% of churn. Moreover, churn is applied each 60 seconds for 5 minutes. The results of the experiments are depicted in Figure 7.

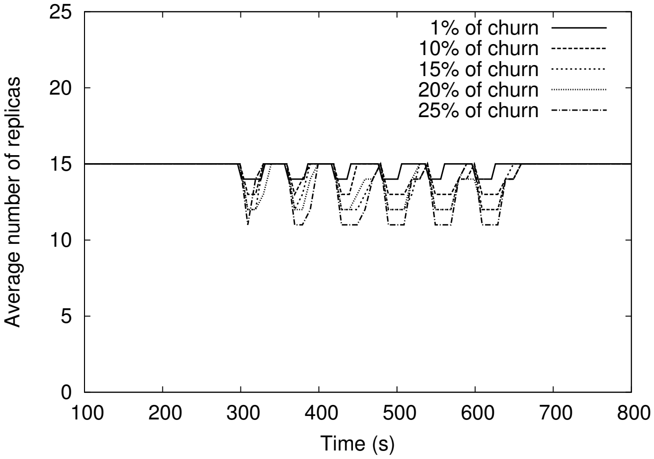


Fig. 7. DATAFLASKS behavior for different levels of churn.

As it is observable, during the churn phase, the average number of replicas is not constant as it is impaired by the departure of nodes. However, DATAFLASKS is able to repair the number of replicas even for churn as high as 25%.

VII. RELATED WORK

Early work on epidemic protocols was applied to replicated database management [23]. Such work is intrinsically related with ours as DATAFLASKS is built solely based on epidemic / gossip protocols. However, our epidemic store is designed to cope with high levels of churn and massive scale systems. Full replication of [23] impedes its applicability to these scenarios.

PAST [36] and OceanStore [37] are large scale storage systems that resemble our DATAFLASKS store for the use of a peer-to-peer, self-organizing overlay network for routing and data discovery. PAST provides a simpler, similar to ours, data model while OceanStore provides some transactional guarantees. Both clearly differ from DATAFLASKS by using structured overlays (Pastry [38] and Tapestry [39] respectively). Our data store is solely based on unstructured peer-to-peer protocols.

Classical relational database management systems offer strong data consistency models that require coordination protocols for distribution. These protocols are known to scale only to a few dozens of nodes [40]. Following this observation distributed storage systems have been the focus of intense research work where data consistency is relaxed privileging scalability properties. Several large scale storage systems were proposed offering these relaxed consistency models which are similar to the key-value data model used in DATAFLASKS. Bigtable [5], Dynamo [3], Cassandra [2] and PNuts [4] are the most prominent examples of these data stores. Although these data stores are suitable for a number of tasks and are able to scale to hundreds of nodes they still rely, similarly to PAST and OceanStore, on a Distributed Hash Table (DHT) such as Chord [41] or variants. The exception being Bigtable which in turn requires a set of master nodes, which is still impractical in very large scale scenarios. Relying on structured overlays impairs their ability to cope with high levels of churn, which are characteristic of DATAFLASKS target environments. In fact, [7] shows that current implementations of DHTs struggle for churn rates observable in real peer-to-peer systems. Furthermore, such work notes that reactive response to churn is insufficient and proposes a periodic, proactive approach for dealing with system dynamics. We argue that using proactive unstructured overlays is the suitable approach in these high churn, highly dynamic scenarios. Additionally, [42] shows that proactive approaches to data replication allow systems to avoid undesirable network spikes that reactive approaches can create after detecting some failure.

In this paper we propose a group construction algorithm. A class of gossip-based protocols that similarly allow to group nodes into groups is distributed slicing [14]–[17]. Slicing protocols could be used for data distribution and replication in DATAFLASKS. However, with these protocols it is not possible to control directly the group size. It is possible to define the group size as a proportion of the total system size but this is impractical. Note that, in the target scenario, knowing the total system size is unattainable. This renders controlling of the replication factor virtually impossible using slicing techniques.

VIII. DISCUSSION

In this paper we presented DATAFLASKS, an epidemic data store for very large scale environments. In order to build DATAFLASKS we designed a new gossip-based protocol, completely decentralized, that autonomously organizes several thousands of nodes into groups. Moreover, it organizes nodes into groups of parameterizable size without the need to have nodes knowing this size. The protocol is designed to integrate with DATAFLASKS minimizing data transfer between nodes when replication levels need to be repaired. Additionally, we sketch a proof of correctness of the algorithm as well as simulation results that validate its convergence properties.

With the group construction algorithm validated we evaluate DATAFLASKS as a whole. We provide a complete DATAFLASKS open source prototype, which can be run both on top of the Minha framework [35] as well as in a real environment. Experiments with DATAFLASKS show that data persistence is guaranteed even in the presence of high levels of churn. We argue that our results support a proactive approach to data replication in very large scale scenarios.

DATAFLASKS opens some interesting research paths for future work. Firstly, any client application of our data store relies on a load balancer component for node discovery. The implementation of this load balancer is an interesting challenge. In fact, the load balancer should provide the client with judiciously chosen nodes to increase the performance of the data store. However it needs to do so still relying only on partial knowledge of the system raising some design challenges. Specially taking into account highly dynamic systems. Secondly, DATAFLASKS relies on a number of configuration parameters. Researching autonomous configuration of the data store is a research path worth pursuing. Finally, on the practical side, it is important to evaluate the performance of DATAFLASKS when compared with other key-value stores.

ACKNOWLEDGMENT

This work is partially financed by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness), by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) with grant SFRH/BD/71476/2010 and by Project Smartgrids - NORTE-07-0124-FEDER-000056, co-financed by the North Portugal Regional Operational Programme (ON.2 – O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

REFERENCES

[1] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. USENIX, 2007.

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

[3] 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 *The ACM Symposium on Operating Systems Principles*. ACM, 2007.

[4] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," in *The International Journal on Very Large Data Bases*. Springer Verlag, 2008.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: a distributed storage system for structured data," in *The Symposium on Operating Systems Design and Implementation*. USENIX, 2006.

[6] N. Leavitt, "Will NoSQL databases live up to their promise?" in *Computer*. IEEE, 2010.

[7] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, "Handling churn in a DHT," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX, 2004.

[8] F. Maia, M. Matos, R. Vilaca, J. Pereira, R. Oliveira, and E. Riviere, "Dataflasks: An epidemic dependable key-value substrate," in *The International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology*. IEEE, 2013.

[9] E. Rivière and S. Voulgaris, "Gossip-based networking for internet-scale distributed systems," in *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2011.

[10] S. Voulgaris, D. Gavidia, and M. V. Steen, "CYCLON: Inexpensive membership management for unstructured p2p overlays," in *Journal of Network and Systems Management*. Springer, 2005.

[11] 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*. Springer Verlag, 2001.

[12] F. Maia, M. Matos, J. Pereira, and R. Oliveira, "Worldwide consensus," in *IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer Verlag, 2011.

[13] P. Jesus, C. Baquero, and P. S. Almeida, "Fault-tolerant aggregation for dynamic networks," in *IEEE Symposium on Reliable Distributed Systems*. IEEE, 2010.

[14] A. Fernandez, V. Gramoli, E. Jimenez, A.-M. Kermarrec, and M. Raynal, "Distributed Slicing in Dynamic Systems," in *The International Conference on Distributed Computing Systems*. IEEE, 2007.

[15] V. Gramoli, Y. Vigfusson, K. Birman, A.-M. Kermarrec, and R. van Renesse, "Sliver, A fast distributed slicing algorithm," in *Symposium on Principles of Distributed Computing*. ACM, 2008.

[16] 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*. Springer Verlag, 2012.

[17] ——, "Slicing as a distributed systems primitive." in *6th Latin-American Symposium on Dependable Computing*. IEEE, 2013.

[18] M. Matos, V. Schiavoni, E. Rivière, P. Felber, and R. Oliveira, "LayStream: composing standard gossip protocols for live video streaming," in *The International Conference on Peer-to-Peer Computing*. IEEE, 2014.

[19] R. Vilaça, R. Oliveira, and J. Pereira, "A correlation-aware data placement strategy for key-value stores," in *11th IFIP International Conference on Distributed Applications and Interoperable Systems*. Springer Verlag, 2011.

[20] R. Vilaça, F. Cruz, and R. Oliveira, "On the expressiveness and trade-offs of large scale tuple stores," in *The International Symposium on Distributed Objects, Middleware, and Applications*. Springer Verlag, 2010.

[21] R. Vilaça and R. Oliveira, "Clouder: a flexible large scale decentralized object store: architecture overview," in *Proceedings of the Third Workshop on Dependable Distributed Data Management*. ACM, 2009.

[22] M. Matos, R. Vilaça, J. Pereira, and R. Oliveira, "An epidemic approach to dependable key-value substrates," in *The International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology*. IEEE, 2011.

[23] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. ACM, 1987.

[24] P. Eugster, R. Guerraoui, A. Kermarrec, and L. Massoulié, "From epidemics to distributed computing," in *Computer*. IEEE, 2004.

[25] A.-M. Kermarrec and M. Van Steen, "Gossiping in distributed systems," in *ACM SIGOPS Operating Systems Review*. ACM, 2007.

[26] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec, "Lightweight probabilistic broadcast," in *Acm Transactions on Computer Systems*. ACM, 2003.

[27] A.-M. Kermarrec, L. Massoulié, and A. J. Ganesh, "Probabilistic reliable dissemination in large-scale systems," in *IEEE Transactions on Parallel Distributed Systems*. IEEE, 2003.

[28] P. Felber, A.-M. Kermarrec, L. Leonini, E. Rivière, and S. Voulgaris, "Pulp: An adaptive gossip-based dissemination protocol for multi-source message streams," in *Peer-to-Peer Networking and Applications*. Springer US, 2012.

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

[30] M. Jelasity, R. Guerraoui, A. Kermarrec, and M. van Steen, "The peer sampling service: Experimental evaluation of unstructured gossip-based implementations," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*. Springer Verlag, 2004.

[31] S. Voulgaris, M. Jelasity, and M. van Steen, "A robust and scalable peer-to-peer gossiping protocol," in *Proceedings of the Second international conference on Agents and Peer-to-Peer Computing*. Springer Verlag, 2005.

[32] R. Guerraoui, R. Oliveira, and A. Schiper, "Stubborn Communication Channels," EPFL, Tech. Rep., 1998.

[33] S. Voulgaris and M. van Steen, "Vicinity: A pinch of randomness brings out the structure," in *Middleware*. ACM/IFIP/USENIX, 2013.

[34] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud Computing*. ACM, 2010.

[35] N. A. Carvalho, J. a. Bordalo, F. Campos, and J. Pereira, "Experimental evaluation of distributed middleware with a virtualized java environment," in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM, 2011.

[36] P. Druschel and A. Rowstron, "Past: a large-scale, persistent peer-to-peer storage utility," in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001.

[37] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: An architecture for global-scale persistent storage," in *SIGPLAN Notices*. ACM, 2000.

[38] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. ACM, 2001.

[39] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz, "Tapestry: A resilient global-scale overlay for service deployment," in *Journal on Selected Areas in Communications*. IEEE, 2006.

[40] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *ACM SIGMOD International Conference on Management of Data*. ACM, 1996.

[41] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup protocol for internet applications," in *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2001.

[42] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz, "Proactive replication for data durability," in *The International Workshop on Peer-to-Peer Systems*. USENIX, 2006.