DATAFLASKS: an epidemic dependable key-value substrate

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*—Recently, tuple-stores have become pivotal structures in many information systems. Their ability to handle large datasets makes them important in an era with unprecedented amounts of data being produced and exchanged. However, these tuple-stores typically rely on structured peer-to-peer protocols which assume moderately stable environments. Such assumption does not always hold for very large scale systems sized in the scale of thousands of machines. In this paper we present a novel approach to the design of a tuple-store. Our approach follows a stratified design based on an unstructured substrate. We focus on this substrate and how the use of epidemic protocols allow reaching high dependability and scalability.**

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

I. INTRODUCTION

Nowadays some of the most interesting challenges in computing deal with large scale systems. Nowadays, processors are not getting faster at the same rate of previous years, instead it is possible to have more of them [1]. This scenario makes it possible to consider data centers with thousands of machines. However, the development of services and software systems that actually take advantage of large scale systems is not a trivial task.

With the exponential growth in the amount of data being produced and exchanged, several approaches were made to build novel tuple-stores able to scale and take advantage of larger resource pools [2]–[5]. Although these data stores proved to be suitable and efficient for a number of tasks [6], they still rely on a structured peer-to-peer systems, typically on a distributed hash table (DHT) or a variant of a distributed hash table. Relying on distributed hash tables requires assuming of moderately stable environment in order to guarantee the overlay availability. However, as the system size grows, the assumption of a moderately stable environment becomes unrealistic. When reaching unprecedented number of nodes, faults and churn become the rule instead of the exception.

We posit that an unstructured but resilient approach to data management is more appropriate in the context of such large-scale systems. In [7], we proposed a novel two-layer approach to the design of a key-value data store. This two-layer approach separates client interface and concurrency control (top layer) from the actual data storage (bottom layer). The proposal is to follow a completely decentralized and unstructured approach to the design of the latter. In this paper we focus on the bottom layer describing DATAFLASKS: an highly resilient and dependable data storage system aimed at very large scale environments.

Along the paper, we present the design of DATAFLASKS and describe the main challenges of such an approach. We also address some of these challenges with concrete solutions and point out our research path to tackle the remaining. We also evaluate the current version of DATAFLASKS in order to validate its scalability.

The remaining of the paper is as follows. Section II provides some background related with epidemic protocols. In Section III we describe the context in which DATAFLASKS appears and its requirements. Section IV presents DATAFLASKS design and Section V its architecture. Evaluation appears in Section VI. We describe open challenges and future steps in Section VII.

II. BACKGROUND

The fact that we are targeting very large scale systems raises a number of interesting challenges. Notably, in very large scale systems any kind of global knowledge is unattainable. Any system that relies on information that grows linearly with system size does not scale and, therefore, is unusable in such a scenario. As a consequence, large scale protocols must solely rely on partial information about the system and on node-local decisions. Fortunately, there is a class of well studied protocols that meet such requirements: *epidemic* or *gossip-based* protocols.

A typical epidemic protocol operates as follows. Each node has a set of neighbors, called its *view*. The protocol progresses by having each node periodically exchanging knowledge with one or several of its neighbors.

The first problem to address is how to maintain the neighbor list refreshed even considering that nodes can become disconnected or simply leave the system. Besides maintaining the *view* refreshed it is important that it exhibits some properties. In particular, epidemic protocols benefit from views composed by a uniformly random sample of nodes [8]. If the *view* is, in fact, a random sample of nodes, choosing a random peer from such list is equivalent to choose randomly from all the nodes in the system. The problem of providing random views of nodes in the system falls in a well studied problem in large scale distributed systems that has been addressed by a family of protocols known as the *Peer Sampling Service*. These protocols are gossip protocols themselves. In a nutshell, each node keeps a set of nodes it knows. Periodically, it refreshes such set by contacting one ore more of those nodes and exchanging information. Notably, this apparently simple approach allows these protocols to provide each node with a random stream of uniformly sampled nodes. Examples of these protocols are Cyclon [9] and Newscast [10].

Interestingly, the collection of views generated by the Peer Sampling Service not only serves as the support for other epidemic protocols but also as an information dissemination medium. From early work on random graphs [11], we know that it is possible, with arbitrarily high probability, to effectively disseminate data in a epidemic fashion provided that each node relays a sufficient number of messages. In particular, taking N as the number of nodes, each node must relay $ln(N) + c$ messages to have a probability of atomic *infection* of $p_{atomic} = \epsilon^{-\epsilon^{-c}}$. Considering *views* of such size, uniformly sampled from the all set of nodes, an overlay emerges that allows for epidemic data dissemination. Note that these views do not grow linearly with system size and are, therefore, a scalable dissemination mechanism. This dissemination mechanism serves as the base for request dissemination in DATAFLASKS.

To complete our background section we should mention distributed systems slicing techniques. Slicing a large-scale distributed system is the process of autonomously partitioning its nodes into k groups, named slices based on some criteria. For instance, it is possible to slice the system according to node storage space or their uptime. Previous work [9], [12]–[17], showed that it is possible to achieve such goal without any kind of global knowledge using epidemic approaches. These slicing protocols rely on a Peer Sampling Service and notifies each node of the slice it belongs to. Slicing will be important for the design of DATAFLASKS.

III. STRATUS

In [18] we proposed a novel massive data store system. In this paper, we will refer to such system as STRATUS. STRATUS is a stratified system where a clear separation of concerns between layers is of crucial importance. It is this separation of concerns that allows this design to scale to thousands of nodes.

The architecture overview of STRATUS is presented in Figure 1. Although the present paper will focus on the bottom layer of STRATUS it is important to provide some context. In particular, to define the set of assumptions supporting such layer.

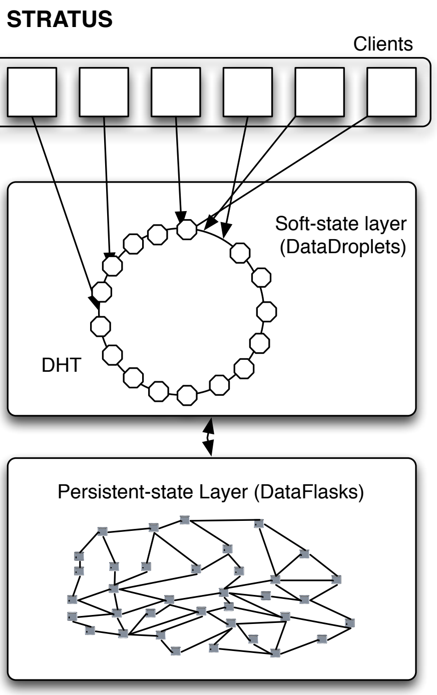
STRATUS
Clients
Soft-state layer (DataDroplets)
DHT
Persistent-state Layer (DataFlasks)

Figure 1. STRATUS Architecture.

Similarly to traditional relational database management systems, in our proposal there is a clear separation between i) client interface and concurrency control concerns and ii) data storage itself. The first set of concerns is addressed by the STRATUS top layer named DATADROPLETS. This upper layer provides 1) client interface, 2) caching, 3) concurrency control, and 4) high level processing. As described in [7], even though DATADROPLETSis itself decentralized, we assume this layer to run mainly in memory and in a moderate size environment. This allows DATADROPLETS to be based on a structured approach. The main challenges of this layer have been addressed [19], [20].

Note that DATADROPLETS has some key characteristics that yield specific responsibilities to the underlying DATAFLASKS. Firstly, its structured approach (aimed at a moderately sized environment) delegates large scale concerns to the bottom layer. Secondly, it is a soft-state layer meaning that, in case of catastrophic failure, it should be possible to reconstruct it completely from the persistent-state layer. This implies that DATAFLASKS must provide data persistence and high availability. Finally, it is responsible for concurrency control and client interface. In our design, this means that DATAFLASKS does not need to take into account conflicts arising from concurrent operations. In fact, DATADROPLETS is responsible for correctly ordering requests, which, in our design, is done by attaching version stamps to every object.

A major requirement for DATAFLASKS is data availability.

Since this layer targets massive scale deployments it should be able to achieve data availability even in highly unstable environments. With this aim in mind, our proposal is to take advantage of high scalability and resilience of epidemic protocols. These have been used previously to build several webscale systems and applications [21] like overlay construction and maintenance [9], [22], consensus [23], data aggregation [24] and distributed slicing [12], [15], [16].

As a result of this design, DATAFLASKS serves as a persistence layer. It must store and serve objects addressed by an identifier following an API consisting of *get(key)* and *put(key, value)* operations. Each object is an array of arbitrary bytes and the only assumption we make about the upper layer is that operations are ordered. In our design this is done by having each object carry a version.

IV. DATAFLASKS

In this Section we introduce the design of DATAFLASKS. We consider a set of nodes, on the order of thousands, where anyone can receive requests from the upper layer. We assume simple put and get operations. Put operations on the same item are totally ordered. Get operations specify the requested version of the item. The following challenges remain:

A. How to distribute data;
B. How to route requests;
C. How to replicate data;

A. Data distribution

The main idea behind our solution to data distribution in DATAFLASKS is to disseminate data in an epidemic fashion and have nodes locally decide if they need to store that individual item. To achieve this we divide the system into sets of nodes. Each set will be responsible for storing a subset of the data according to its key range. Slicing mechanisms make it possible to have a system autonomously dividing itself into sets of nodes (slices) without any kind of global knowledge. Moreover, this process can leverage certain locally measured attributes and partition the system according to some criterion. In our case the system will be sliced according to the individual node storage capacity. This allows that a certain node with less capacity is assigned with less data to store. Any other criteria could be used, though.

Slicing protocols provide each node with the slice it belongs to and thus serve as a data distribution mechanism. In addition, these protocols are highly resilient and continuously adapt to changes in the system, let them be caused by faults or churn. Note that, we could simply toss a coin and decide to which slice a node belongs to. Provided we had uniformity on that process it would be enough for partitioning the system. However, such approach is not resilient to correlated faults. For instance, if, for some reason, a significant portion of a certain slice fails there would be no way of effectively recover by coin tossing. On the other hand, the slicing mechanism is able to adjust to such failure and nodes would change slice in order to balance system distribution.

B. Request routing

We have provided a mechanism to divide the system into slices (sets of nodes) and to distribute objects across nodes with the help of such partitioning. It is necessary now for requests to reach the appropriate nodes. A write request must reach the corresponding set of nodes and a read request must reach at least one node holding the target item.

Both the request routing and data dissemination is accomplished with a Peer Sampling Service. As mentioned before, each node can use the *view* provided by the Peer Sampling Service to disseminate information. In order for a node to disseminate a message in the system it suffices to send it to the nodes in its local view. However, it is important to notice that disseminating all requests to all nodes is rather inefficient. In DATAFLASKS, some optimizations are possible which avoid such problems. For instance, atomic dissemination for most requests in not needed. It is sufficient to reach only the percentage of system nodes that guarantees that some nodes of the target slice are reached. Following the ideas described in [17], we consider a Peer Sampling Service intra-slice. Once a request reaches a node in its target slice, dissemination is done only to nodes of that slice.

C. Data replication

To tolerate node failures and churn data is replicated among the nodes of each slide.

Although straightforward, this approach raises some challenges. Namely, with regard to the decision on the number of slices to choose and to their size. Fortunately, recent slicing protocols [17] allow for dynamic configuration of the slicing mechanism. This opens the door to autonomous mechanisms for replication management. Note that, for the same system size, a smaller number of slices increases the replication factor but lowers system capacity. This is a consequence of the limited capacity of the individual node. Each node can replicate a limited number of objects which, in turn, limits the number of objects a slice can hold. Conversely, increasing the replication factor increases the performance of read operations that can target more nodes and increases system capacity. In this regard, we believe that this opens important research paths for future work.

In this work we simply define the size of slices as a percentage of the system size. Furthermore, we make the assumption that the size of the slices is large enough so that the risk of having all nodes within the same slice failing simultaneously is null.

V. DataFlasks Architecture

Figure 2 presents the architecture supporting DataFlasks. Two main components are considered: a DataFlasks host and a client library. These components depend on a set of services each with a specific task.

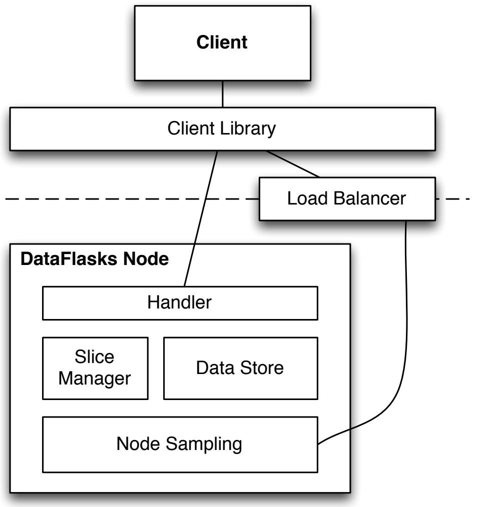

Figure 2. DataFlasks Architecture.

Each DataFlasks node has four services running. A *Slice Manager* service is responsible for partitioning the system slices. As described above, this service is implemented by a slicing protocol. In our case, by DSlead [17]. The *Peer Sampling Service* which is responsible for providing references of other nodes in the system and also to the *Load Balancer* service. The *Load Balancer* in turn, provides the *Client Library* with references to nodes that can answer client requests. The implementation of the *Load Balancer* has a significant impact on the performance of the system. For now, the *Load Balancer* provides the client with a random contact node. Further considerations on this service are made in Section VII. Finally, the request *Handler* is responsible for dealing with requests made to the node. It knows to which slice the node belongs to from the *Slice Manager* and stores and retrieves correspondent data to and from the *Data Store*. The *Data Store* is an abstraction of the actual storing mechanism which can be the node hard disk or other persistence mechanism.

The client library is divided into two subcomponents. One is responsible for implementing the DataFlasks API and serves client requests by contacting a DataFlasks node. The other is responsible for dealing with reply messages. The second component must know how to handle multiple replies for the same request. This happens due to the epidemic dissemination of the requests and the multiple nodes able to reply to them. In order to be able to do so, read requests carry a request identifier in order to distinguish multiple read requests for the same object.

VI. Evaluation

Although DataFlasks is still under development and some challenges remain to be tackled, in this section we provide an early evaluation of the prototype. The experiments we present serve as a motivation to continue the design and implementation of our epidemic key-value store substrate. More specifically, we try to verify if DataFlasks can effectively scale to thousands of nodes.

In these experiments we use *Minha* [25] as the simulation environment. *Minha* is a event driven simulation framework that has the key advantage of allowing to run real, unmodified and not instrumented, Java application code. Both DataDroplets and DataFlasks are implemented in Java.

For the first experiment, we ran DataFlasks configured to consider ten slices and an increasing number of nodes. At this stage of development we ran DataFlasks decoupled from DataDroplets and used YCSB [26] cloud storage benchmark as its direct client. We ran YCSB configured for a write only workload. For each run we measured the average number of messages each node had to send/receive to perform the YCSB requests. In Figure 3 we present the results of this experiment.

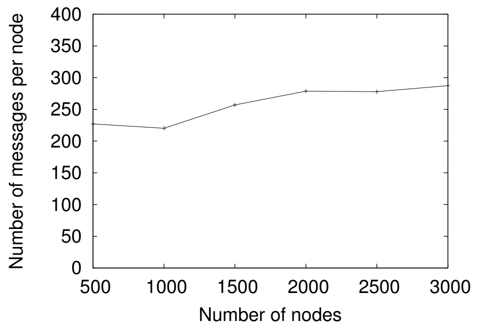

Figure 3. Average number of messages per node with constant number of slices.

Notice that despite the increase in the number of nodes from 500 to 3,000 the number of messages handled by each node remains roughly the same. Considering that we are keeping the number of slices constant, this means that to significantly increase the replication factor it suffices to add more nodes.

For the second experiment we increased the number of slices proportionally to the increase in the number of nodes. This renders the replication factor constant. While in the first scenario the increase in the number of nodes privileged a greater replication factor, in this second experiment the extra number of nodes is used to enlarge the system capacity. Figure 4, depicts the second experiment results.

Similarly to the previous experiment, the number of messages handled by each node increases gracefully. Indeed, it shows a sub-linear variation with respect to the number of

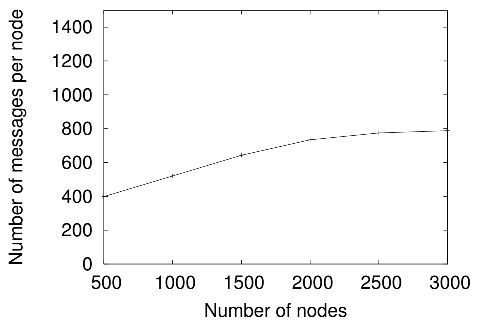

Figure 4. Average number of messages per node with variable number of slices.

nodes. The difference in the number of messages between the two experiments is explained by how the *Load Balancer* is used. As each operation is requested to randomly chosen node and, in the second experiment, we are increasing the number of slices, the probability of immediately reaching a node of the intended slice is increasingly lower. This leads to a larger number of messages being disseminated.

Both results hint at promising scalability properties of DATAFLASKS.

VII. DISCUSSION

Along this paper we presented the main ideas behind the design of an epidemic key-value substrate. We provided mechanisms to partition the data across nodes in the system, to disseminate requests to appropriate nodes and to have objects replicated through multiple locations. However, some key aspects of the system are still missing and are object of ongoing research. These are mainly: maintaining replication level in face of churn or faults and optimizing request dissemination.

Concerning the maintenance of replication levels the problem is that, in our design, there is no centralized way of knowing if every object has, in fact, at least r replicas. Notice that, as described earlier, each slice stores a set of data objects. If the slice size is greater than r the replication factor is assured. However, in order to provide persistence guarantees extra assumptions must be made. For instance, we have to assume that, for each slice, there are always some correct number of nodes. Otherwise, it seems impossible to guarantee that a certain object is not lost. These assumptions are still to be further analyzed and studied. Moreover, the case where a node joins or leaves a certain slice is still to be treated appropriately. Such operations can have a serious impact in performance and persistence guarantees if implemented improperly. For instance, it must be enforced that a situation where every node in a certain slice decides to change slice never happens. At the same time, when a node joins a certain slice, mechanisms for efficient state transfer must be devised. Otherwise, we may fall into a situation where the majority of the system is concerned with state transfer at the expense of actual data request processing.

Another important research path is related to system optimization. In DATAFLASKS, the client library, along with the *Load Balancer*, are components where various optimizations can be studied. These are mainly related to node discovery and the decision to which node route each request. The intuition behind this idea is simple. If the *Load Balancer* was able to know exactly which node to contact for each request, dissemination mechanisms would be reduced to the minimum. As this is not feasible in practice, cache mechanisms should be studied in order to achieve a solution as close as possible to the ideal one.

ACKNOWLEDGMENT

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 projects STRATUS/FCOMP-01-0124-FEDER-015020 and FCOMP-01-0124-FEDER-022701.

REFERENCES

[1] H. Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005. [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," in *SIGOPS Oper. Syst. Rev.* 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 *SOSP '07*. 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 *VLDB*. VLDB Endowment, 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 *OSDI '06*. USENIX Association, 2006.

[6] N. Leavitt, "Will nosql databases live up to their promise?" *Computer*, vol. 43, no. 2, pp. 12–14, Feb. 2010. [Online]. Available: http://dx.doi.org/10.1109/MC.2010.58

[7] M. Matos, R. Vilaça, J. Pereira, and R. Oliveira, "An epidemic approach to dependable key-value substrates," in *DCDV'11*. IEEE, 2011.

[8] S. Voulgaris and M. Steen, "Epidemic-style management of semantic overlays for content-based searching," in *Euro-Par 2005 Parallel Processing*, ser. Lecture Notes in Computer Science, J. Cunha and P. Medeiros, Eds. Springer Berlin Heidelberg, 2005, vol. 3648, pp. 1143–1152. [Online]. Available: http://dx.doi.org/10.1007/11549468_125

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

[10] S. Voulgaris and e. al, “A robust and scalable peer-to-peer gossiping protocol,” *Proceedings of the Second international conference on Agents and Peer-to-Peer Computing*, 2005. [Online]. Available: http://www.springerlink.com/index/ey3aehdf7j5uauqt.pdf

[11] P. Erds and A. Rnyi, “On the evolution of random graphs,” in *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, 1960, pp. 17–61.

[12] 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.

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

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

[15] 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.

[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*, 2012.

[17] ——, “Slicing as a distributed systems primitive.” in *6th Latin-American Symposium on Dependable Computing (LADC’13)*, 2013.

[18] R. Vilaça and R. Oliveira, “Clouder: a flexible large scale decentralized object store: architecture overview,” in *WD-DDM ’09: Proceedings of the Third Workshop on Dependable Distributed Data Management*. New York, NY, USA: ACM, 2009, pp. 25–28.

[19] 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.

[20] R. Vilaça, F. Cruz, and R. Oliveira, “On the expressiveness and trade-offs of large scale tuple stores,” in *On the Move to Meaningful Internet Systems, OTM 2010*. Springer Berlin / Heidelberg, 2010, vol. 6427, pp. 727–744.

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

[22] 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.

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

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

[25] N. A. Carvalho, J. a. Bordalo, F. Campos, and J. Pereira, “Experimental evaluation of distributed middleware with a virtualized java environment,” in *MW4SOC ’11*. ACM, 2011.

[26] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *SoCC*, 2010.