FogStore: Toward a Distributed Data Store for Fog Computing

Ruben Mayer
Institute of Parallel and Distributed Systems
University of Stuttgart, Germany
Email: ruben.mayer@ipvs.uni-stuttgart.de

Harshit Gupta, Enrique Saurez, Umakishore Ramachandran
Georgia Institute of Technology
Atlanta, Georgia, USA
Email: {harshitg, esaurez, rama}@gatech.edu

Abstract—**Stateful applications and virtualized network functions (VNFs) can benefit from state externalization to increase their reliability, scalability, and inter-operability. To keep and share the externalized state, distributed data stores (DDSs) are a powerful tool allowing for the management of classical trade-offs in consistency, availability and partitioning tolerance. With the advent of Fog and Edge Computing, stateful applications and VNFs are pushed from the data centers toward the network edge. This poses new challenges on DDSs that are tailored to a deployment in Cloud data centers. In this paper, we propose two novel design goals for DDSs that are tailored to Fog Computing: (1) Fog-aware replica placement, and (2) context-sensitive differential consistency. To realize those design goals on top of existing DDSs, we propose the FogStore system. FogStore manages the needed adaptations in replica placement and consistency management transparently, so that existing DDSs can be plugged into the system. To show the benefits of FogStore, we perform a set of evaluations using the Yahoo Cloud Serving Benchmark.**

Index Terms—**Distributed Data Store, Fog Computing, Consistency**

I. INTRODUCTION

Large-scale distributed applications gain increasing importance in almost every aspect of life. Beyond traditional applications, via Network Function Virtualization (NFV), *network functions* transform from dedicated hardware middleboxes to large-scale distributed applications. Fueled by the developments in Software-defined Networking (SDN), Virtualized Network Functions (VNFs) are increasingly prevailing. Most of the applications, SDN controllers, and VNFs are stateful [14], [16]. Statefulness poses challenges on failure recovery and state sharing of applications [4], [18].

As an increasing trend, applications and VNFs are being deployed in Cloud data centers, which offer virtually unlimited resources, high availability, and reduced administration complexity. The recent trend of Fog Computing [13], [8], [17] foresees nodes with computational and storage capabilities to be placed close to the edge of the network. The network of Fog nodes builds a computational continuum between the end users' devices and the Cloud data centers. Different from Cloud Computing, Fog nodes are not necessarily deployed in data centers; instead, hardened routers, cellular access points,

and smart home gateways also participate in the computational continuum. When Fog nodes are deployed at the edge of the network, they can exploit the *locality* of clients, applications, and data, resulting in reduced latency and network load. The ongoing trend will push many applications and VNFs out from the central Cloud data centers toward the Fog.

Stateful applications benefit from *state externalization*, i.e., exposing their internal state to other applications, for *failure recovery* and *state sharing* [4], [18]. Replicating and sharing state between different processes is a complex endeavor with many challenges and pitfalls, e.g., keeping consistency between the replicas and supporting concurrent read and write access. Hence, to manage externalized state, highly available *distributed data stores* (DDSs) are often employed. A DDS keeps multiple replicas of the stored data records on different physical nodes. In doing so, DDSs handle the complex trade-off between consistency, availability, latency and partitioning tolerance [3], [1], [19]. To benefit from Fog Computing, DDSs must be pushed from the Cloud to the Fog infrastructure.

However, the design of current DDSs—that are designed for Cloud data centers—builds on assumptions that do not hold true for Fog Computing. First, the replica placement strategies employ *data center failure models*, i.e., mask failures that typically happen in a data center. For that reason, *rack-aware* placement algorithms, that avoid placing multiple replicas of the same data on the same server rack, are predominant. However, such a placement is not generally applicable to Fog Computing, where we assume a more heterogeneous infrastructure that does not always provide classical server racks. Second, current DDSs do not take into account context, especially, locality of the clients; instead, all clients are provided with the *globally same* read and write consistency guarantees. The assumption of uniform consistency requirements does not always hold true for Fog Computing applications. Instead, we observe, that often, the *context*, e.g., the location, of clients and data determines the consistency requirements.

To tackle the shortcomings of existing DDSs with regard to Fog Computing, in this paper, we propose a Fog-enabled DDS abstraction—called **FogStore**—that implements two novel design goals: *Fog-aware replica placement* and *context-sensitive differential consistency*. FogStore manages the needed adaptations in replica placement and consistency management transparently, so that existing made-for-cloud
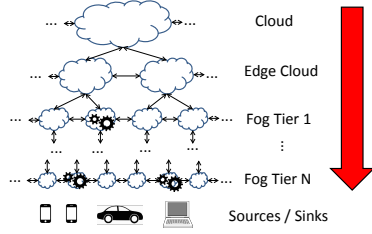
Fig. 1. The Fog continuum provides a hierarchical network of nodes with computational and storage capabilities.
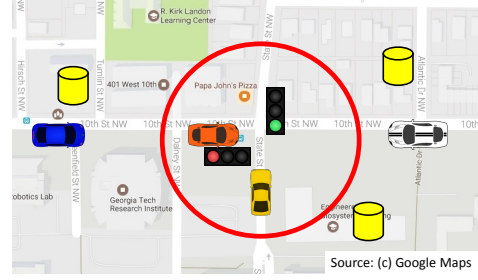

Fig. 2. Differential consistency: only cars close to the traffic lights (within the red circle) need to read consistent data about the traffic light state.

DDSs can be plugged into the system. Further, we evaluate the impact of the design goals of FogStore on the overall system performance based on a data store benchmark.

II. SYSTEM MODEL

Here, we introduce a system model of Fog Computing infrastructure along with an application and DDS model.

A. Fog Computing Model

Fog Computing describes a computational continuum between the traditional cloud data centers and the data sources and sinks [8], [17]. It consists of an orchestrated set of *Fog nodes* that are typically hierarchically organized in multiple tiers [13], [20], as depicted in Figure 1. In the computational continuum of Fog Computing, a Fog node is a distinct host that provides computational and storage capabilities (denoted as the *Fog platform*) and runs a software stack for deployment of applications and management and interaction between Fog nodes (denoted as the *Fog software*) [8]. The computational and storage capabilities of the Fog platforms can be heterogeneous. The Fog software provides an abstraction layer for the deployment of applications and services on the Fog node using virtualization technology such as containers.

In this paper, we assume that Fog nodes can fail according to the fail-recovery model, where an arbitrary number of nodes may fail and restart at any time. The connection between Fog nodes can be interrupted or delayed. Hence, sets of Fog nodes can become temporarily incapable of communicating with each other, i.e., temporary network partitioning is possible.

B. Application and Data Store Model

We assume *stateful* applications that have an inherent *locality* of their data sources and sinks, meaning that data sources and sinks are located in the same physical region. Further, we assume that the applications can expose their internal state to a DDS and read state stored in a DDS for failure recovery and state sharing between different application instances.

Multiple instances of a DDS are deployed across the Fog Computing continuum. The data to be stored is replicated across the DDS instances—the copies of a specific data record are referred to as *replicas*. In doing so, the DDS faces the problem of partitioning the data, placing the replicas of the data partitions on the available DDS instances, and keeping consistency between the replicas.

We assume that the DDS is able to support individual *consistency levels* for each single read or write operation on the

data. A consistency level specifies how many of the replicas of a data set need to be retrieved or updated until the operation is reported as completed to the querying client. For instance, a *read consistency* level ONE returns a read result to the client when the data has been retrieved from one single replica, whereas a *write consistency* level QUORUM would return a write query as successful to the client only after a majority of data replicas has been updated. Modern DDSs, such as Apache Cassandra [15], allow for a fine-grained specification of the consistency level on each single (read or write) operation.

III. FOGSTORE

To overcome the shortcomings of existing Cloud DDSs with regard to Fog Computing, we propose the FogStore system. FogStore allows for plugging in existing made-for-Cloud DDSs, extending them with Fog-aware replica placement strategies and context-sensitive differential consistency capabilities that exploit client and data locality in Fog Computing. In the following, we explain the design goals and algorithms of FogStore in more detail.

A. Design Goals

1) Fog-aware Replica Placement: For Cloud data centers, the placement problem has been tackled by rack-aware placement strategies that avoid to place multiple replicas of the same data in the same server rack. However, in Fog Computing, we do not assume that a placement in server rack is always appropriate or possible. The failures that can happen in Fog Computing are different from Cloud Computing. For instance, network partitioning can become a large problem in Fog Computing. While a Cloud data center is usually connected via redundant links to the rest of the network, in Fog Computing, whole groups of Fog nodes can be connected to the rest of the network over a single link—which might even be a wireless link. Instead of mainly focusing on server racks as the standard *failure group* (i.e., a group of nodes failing together because of the same technical reason), a placement strategy for Fog Computing must also take into account the network topology and heterogeneity of the Fog nodes.

Another common assumption of Cloud Computing is that the latency between different nodes of a Cloud data center is negligible. Typically, a customer of a Cloud service does not have full control over the placement of her virtual machines in the physical infrastructure. Contrary to that, in
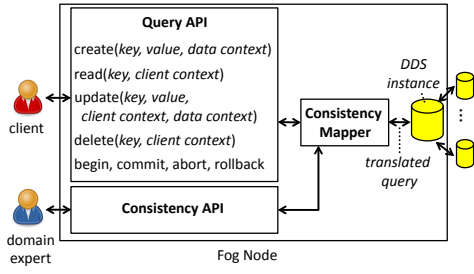
Fig. 3. FogStore architecture.

Fog Computing, latency between Fog nodes is a *first-class citizen*; the placement of replicas in a Fog infrastructure plays a major role in the DDS performance, as also our evaluations in Section IV confirm. A Fog-enabled replica placement strategy should optimize the placement for achieving minimal latency in between the replicas and between the replicas and the data sources and sinks (i.e., the clients).

2) Context-Sensitive Differential Consistency: In Cloud Computing, a common assumption is that clients of a DDS are geo-distributed, concurrently accessing the same data from different locations around the globe. Hence, each client gets provided with the same consistency guarantees on its (write and read) operations on the DDS. Contrary to that, in Fog Computing, different clients accessing the distributed data store often have an individual *context*. This context can influence their requirements on consistency. FogStore allows to exploit context, in particular location, by providing a mapping from a client's context to the consistency level of the client's query. To illustrate this, we discuss an example of differential consistency, using location context, in the following.

In a situation-aware application, multiple autonomous cars read the state of a traffic light on a specific road junction (cf. Figure 2). The cars need strong read consistency when they are close to the junction (i.e., within the red circle): No two cars should read contradicting traffic light status from different DDS nodes. This means that each car has to read the status from all nodes, and use the freshest one to make its decision. However, cars further away from the junction might use the traffic light status for non-critical operations such as to adjust their speed in order to save energy or to update the estimated arrival time in their navigation system. Two different cars further away from a given traffic light may read contradicting traffic light status without inducing safety risks. Hence, those cars can read from one local node, making the response faster and reducing the load on the Fog nodes.

In the following, we describe how the architecture of FogStore supports the proposed Fog-aware design goals.

B. Architecture and Algorithms

FogStore is an extension to existing DDSs that allows for their seamless integration into a Fog Computing environment. Each instance of the DDS, deployed on a Fog node, is plugged into an instance of FogStore. The FogStore instance receives the clients' queries and translates them to DDS queries, implementing Fog-aware replica placement and

context-sensitive differential consistency. FogStore consists of three main components, as depicted in Figure 3: (1) A Query API to receive the create, read, update, delete and transactional queries from the DDS clients, enriched with context information. (2) A Consistency API to receive specifications of *consistency regions* from a domain expert who knows how client context influences the consistency requirements. (3) A Consistency Mapper that, based on the specifications in the Consistency API, automatically maps an incoming query to an appropriate consistency level and issues a corresponding query in the plugged-in DDS. Replica placement is also handled in this component.

1) Query API: Queries are sent by the clients to the Query API of FogStore. Based on the consistency levels specified in the Consistency API, the queries are translated by the Consistency Mapper and forwarded to the underlying DDS.

`create(key, value, data context)` – creates a new key and value. Replicas of the data record are placed on the Fog nodes by the Consistency Mapper according to the *placement algorithm.*

`read(key, client context)` – retrieves the value associated with the key from the DDS.

`update(key, value, client context, data context)` – updates the given key with a new value. By setting the field `data context` in the query, the data context can be updated, e.g., when the data source has moved to a different location. The updated data context is taken into account by the Consistency Mapper in all future queries.

`delete(key, client context)` – deletes the key from the distributed data store.

`tx begin, commit, abort, rollback` – allows for integrating transactional support.

2) Consistency API: In the Consistency API, a domain expert specifies the mapping between data context, client

---

Algorithm 1 FogStore Consistency Mapper

1: DDS localDDS   ▷ links to closest DDS node
2: **procedure** MAPANDEXECUTEQUERY(query, key, client_ctx)
3:      consistencyRegion ←
4:          ConsistencyAPI.`getRegion`(key, client_ctx)
5:      consistencyLevel ←
6:          ConsistencyAPI.`getLevel`(consistencyRegion)
7:      translatedQuery ← `translate`(localDDS.type, query, consistencyLevel)
8:      localDDS.`execute`(translatedQuery)
9: **end procedure**

Algorithm 2 FogStore Replica Placement Algorithm

1: **procedure** PLACEMENT(data_key, data_location, failure_groups, fog_topology, replication_factor)
2:      closestFogNode ← `findClosest`(fog_topology, data_location)
3:      replicaNodes.`add`(closestFogNode)
4:      **while** replicaNodes.size < replication_factor **do**
5:          find nearest neighbor of closestFogNode that is not in same failure group
6:          add it to the replicaNodes
7:      **end while**
8:      `setDataStoreMapping`(key, replicaNodes)
9: **end procedure**

context, and consistency level of the different kind of queries. For location context, which is a major class of context in Fog Computing, the Consistency API provides an easy-to-use abstraction, called *consistency regions*. Note, that beyond location context, other types of (application-specific) context can be extended to the user's needs. In the following, we describe the concept of consistency regions in more detail.

Consistency regions provide information about the consistency levels needed in specific geographical regions around the data location. Those specifications are usually application-specific, i.e., each application using FogStore can have individual consistency regions specified. For instance, in the traffic application described in Section III-A2, the consistency regions would be specified as: "Read from `ALL` replicas when client is located around 500 meters from the data location (i.e., the traffic light location), else read from `ONE` replica".

3) Consistency Mapper: The Consistency Mapper component uses the consistency regions specified in the Consistency API in order to translate client queries to the Query API into queries to the connected DDS. In doing so, an adapter to the connected DDS has to be provided, such that the consistency levels specified in the Consistency API can be translated according to the query language and consistency capabilities of the DDS.

a) Consistency Mapping Algorithm: Algorithm 1 lists the consistency mapping algorithm. First, the consistency region is determined based on the client's context (line 3). Then, the consistency level of that region is determined (line 5). Finally, the query and its consistency level are translated into the appropriate query language of the underlying DDS and executed (lines 7–8).

b) Replica Placement Algorithm: Whenever a new data record is inserted into the DDS (`create` is called in the Query API), the record and its replicas will be placed on the DDS instances according to a placement algorithm.

The Fog-aware replica placement algorithm applied in Fog-Store is centered around the definition of *failure groups*. A failure group is a group of nodes that will fail for the same technical reason (e.g., a local power outage). This can be a group of nodes on a rack, but also a group of nodes that would be disconnected or partitioned because they access the Fog continuum via the same physical link. The concrete setup of failure groups is configured by a domain expert who knows the physical conditions of the Fog continuum. In FogStore, the domain expert can group arbitrary Fog nodes in failure groups and provide them to the placement algorithm.

Algorithm 2 lists the replica placement algorithm. The algorithm takes into account the location of the data, the failure groups and the Fog topology, and the required replication factor (line 1). It searches for the closest Fog node to the data in the Fog topology (line 2) to place the first replica (line 3). The further replicas are placed on the closest nodes of the first replica that are not in the same failure group (lines 4–7). When all replicas are placed, the mapping of replicas to Fog nodes is enforced on the data store (line 8). The distance between Fog nodes can be determined via network coordinates [10].

IV. EVALUATIONS

We perform micro benchmarks that indicate how latency between replicas and different consistency levels influence the read and write latency in a distributed data store. We hasten to add that these are preliminary results simply to illustrate the concept of FogStore and the impact of its design goals.

A. Evaluation Setup

We deploy the MaxiNet network emulator [22] on a 64 vCPU virtual machine with 128 GB of RAM and Ubuntu 16.04.2 LTS, using the OpenStack platform. The underlying hardware is a Supermicro X8OBN server with 8 x 10 Core Xeon E7 8870@2.2GHz CPUs and 1 TB RAM.

In general, the Fog nodes at a given level of the network hierarchy are assumed to form a peer-to-peer network with sufficient redundancy in connectivity to ward off link failures, node isolation, and network partitioning. However, for the purposes of carrying out controlled experiments, the Fog Computing environment we deploy for our experimental study consists of 6 Fog nodes in a star topology with a switch in the middle. This topology reflects 6 different Fog nodes deployed at different locations at the network edge, which are interconnected via a backbone network represented by the central switch. We decided to use this simplification of real network topologies, because it allows for a straight-forward parameterization to simulate different physical distances between Fog nodes. The first 5 Fog nodes host instances of Apache Cassandra 3.10; the 6th Fog node hosts a Yahoo Cloud Serving Benchmark (YCSB) [9] adapter, serving as the "client" that poses queries on the DDS. We use the YCSB-D ("read latest") core workload that fits well with the traffic scenario from Section III-A2. All evaluation runs employ a replication factor of 5, i.e., each data set is replicated on each Cassandra node.

In the evaluations, we use 3 different *network latency settings*, reflecting different placement qualities. In the *low* network latency setting, the latency between the switch and Cassandra nodes is 4, 5, 6, 7 and 8 ms. In the *medium* network latency setting, the latency between the switch and Cassandra nodes is 8, 10, 12, 14 and 16 ms. In the *high* network latency setting, the latency between the switch and Cassandra nodes is 12, 15, 18, 21 and 24 ms. The latency between YCSB node and switch is 1 ms in all settings.

In each of the network settings, we run the YCSB-D workload with different consistency levels for read and write consistency. (1) Consistency level `ONE`: read/write from one replica. (2) `TWO`: read/write from two replicas. (3) `QUORUM`: read/write from a majority (i.e., three) of replicas. (4) `ALL`: read/write from all five replicas.

B. Results and Discussion

The results of the benchmarks are depicted in Figure 4. The stacked bars represent the percentiles of latency measured per operation: minimal latency, 25th, 50th, 75th, 95th and 99th percentile. We divided the latency measurements into read latency and write latency. When measuring read latency, the
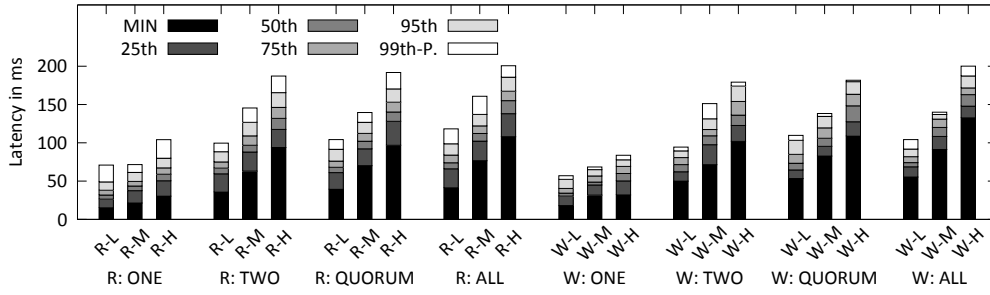
Fig. 4. Read and Write latency on Cassandra for different consistency levels at Low, Medium and High network latency.

consistency level for writes was fixed to `ONE`; when measuring write latency, the consistency level for reads was fixed to `ONE`.

There are two main results. First, the biggest difference in latency is between consistency level `ONE` and any other level. The latency difference between consistency level `ONE` and `TWO` is much higher than the difference between `TWO` and `QUORUM` or `ALL`. Further, consistency level `ONE` does not suffer from high latency penalties when the latency between Fog nodes is higher. This is because consistency level `ONE` does not need coordination between multiple replica nodes, hence, avoiding additional round trips in the network. The coordinator that receives a query can directly execute it locally and return the result to the client.

Second, we see that already small changes in network latency have a high impact on the latencies in higher consistency levels. This is because of the required coordination between replicas, which implies additional network round trips.

From the preliminary studies, we can make some important observations. First, the most beneficial consistency decisions in context-sensitive differential consistency are between consistency level `ONE` and any other high consistency level. Second, if high consistency levels must be enforced, it is particularly important to place the replicas close to each other.

This fits well to the design goals of FogStore. First, the Fog-aware replica placement algorithm treats network latency as a first-class citizen, so that replicas will be placed as close as possible to each other and to the clients. Second, the context-sensitive differential consistency allows for fast responses when clients can deal with reduced consistency in their current context.

V. RELATED WORK

Highly specialized state management protocols can allow for high throughput and low latency, e.g., in VNFs [18], [4]. However, a customized state management for each single application is error-prone and hinders state sharing between multiple different applications. Instead, it is often more practicable to use existing general-purpose distributed data stores. This proposition is in line with a recent industry experience paper from Davie et al. [11], where the authors achieve *interoperability* between different SDN controllers by employing an external data base.

DDSs operate in a trade-off between availability, consistency, latency, and partitioning tolerance. The CAP theorem [3] states that in times of network partitions, the data store

can choose between data consistency and availability, while the PACELC theorem [1] extends CAP, pointing out a tradeoff between consistency and latency in times when the network is not partitioned. Regarding transactional consistency, the available data stores either provide strict ACID properties, or BASE (Basically Available, Soft state, Eventually consistent) [19] properties.

Chun et al. [7] address the question of *how many* replicas of a data record are needed to survive a given failure rate of disks by modeling disk failure and data replication as a birth-death process. The replication degree is an important question to be considered in FogStore as well; however, it is questionable whether failures in a Fog architecture can be modeled in the same way as disk failures in a data center. Doceur and Wattenhofer [12] place a given number of replicas on nodes such that the data availability is maximized; however, they do not take into account communication latency, which is a crucial factor in Fog Computing. The latency-aware replica placement algorithm proposed by Szymaniak et al. [21] is based on the assumption that placing more replicas in a heavily-loaded network region improves the access latency for a large number of clients. This does not apply to Fog Computing scenarios where the locality of clients and data plays a prominent role.

There are approaches to adapt the consistency level of DDSs based on different performance metrics. Harmony [5] is a system that adapts consistency levels based on the probability of stale reads. The Bismar system [6] employs monetary cost as a performance metric for adapting consistency levels. Aslan and Matrawy [2] propose a framework that implements tunable consistency based on arbitrary performance goals of the application. Different from FogStore, all of these systems do not take into account the context of data and clients and only provide a global consistency setting that treats all client requests as a black box.

VI. CONCLUSION

Distributed data stores are an important building block for stateful applications and VNFs. Pushing the data stores to the Fog Computing continuum requires to devise new placement strategies. Further, the context-awareness of Fog Computing scenarios can be utilized in adapting the consistency level to the data and client context. To this end, we propose the FogStore system that provides a Fog-aware replica placement algorithm and context-sensitive differential consistency.

REFERENCES

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, Feb 2012.

[2] Mohamed Aslan and Ashraf Matrawy. Adaptive consistency for distributed sdn controllers. In *2016 17th International Telecommunications Network Strategy and Planning Symposium (Networks)*, pages 150–157, Sept 2016.

[3] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, 2000.

[4] Eleonora Cau, Marius Corici, Paolo Bellavista, Luca Foschini, Giuseppe Carella, Andy Edmonds, and Thomas Michael Bohnert. Efficient exploitation of mobile edge computing for virtualized 5g in epc architectures. In *2016 4th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pages 100–109, March 2016.

[5] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S Pérez. Harmony: Towards automated self-adaptive consistency in cloud storage. In *2012 IEEE International Conference on Cluster Computing*, pages 293–301, Sept 2012.

[6] Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S Pérez. Consistency in the cloud: When money does matter! In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 352–359, May 2013.

[7] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 4–4, Berkeley, CA, USA, 2006. USENIX Association.

[8] OpenFog Consortium. OpenFog Reference Architecture. https://www.openfogconsortium.org/ra/, 2017. [Online; accessed 05-Sep-2017].

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[10] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: A decentralized network coordinate system. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 15–26, New York, NY, USA, 2004. ACM.

[11] Bruce Davie, Teemu Koponen, Justin Pettit, Ben Pfaff, Martin Casado, Natasha Gude, Amar Padmanabhan, Tim Petty, Kenneth Duda, and Anupam Chanda. A database approach to sdn control plane design. *SIGCOMM Comput. Commun. Rev.*, 47(1):15–26, January 2017.

[12] John R Douceur and Roger P Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In *International Symposium on Distributed Computing*, pages 48–62. Springer, Berlin, Heidelberg, 2001.

[13] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[14] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association.

[15] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[16] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol, and Anja Feldmann. Logically centralized?: State distribution trade-offs in software defined networks. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 1–6, New York, NY, USA, 2012. ACM.

[17] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. The fog makes sense: Enabling social sensing services with limited internet connectivity. In *Proceedings of the 2Nd International Workshop on Social Sensing*, SocialSens'17, pages 61–66, New York, NY, USA, 2017. ACM.

[18] Manuel Peuster and Holger Karl. E-state: Distributed state management in elastic network function deployments. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 6–10, June 2016.

[19] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[20] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 258–269, New York, NY, USA, 2016. ACM.

[21] Michal Szymaniak, Guillaume Pierre, and Maarten Van Steen. Latency-driven replica placement. *IPSJ Digital Courier*, 2:561–572, 2006.

[22] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed emulation of software-defined networks. In *2014 IFIP Networking Conference*, pages 1–9, June 2014.