# Technical Report

No. MCC.2019.1

**Cite this report as:**

Jonathan Hasenburg, Martin Grambow, David Bermbach FBase: A Replication Service for Data-Intensive Fog Applications. Technical Report MCC.2019.1. TU Berlin & ECDF, Mobile Cloud Computing Research Group. 2019.

**Abstract:**

The combination of edge and cloud in the fog computing paradigm enables a new breed of data-intensive applications. These applications, however, have to face a number of fog-specific challenges which developers have to repetitively address for every single application.

In this paper, we propose a replication service specifically tailored to the needs of data-intensive fog applications that aims to ease or eliminate challenges caused by the highly distributed and heterogeneous environment fog applications operate in. Furthermore, we present our prototypical proof-of-concept implementation FBase that we have made available as open source.

# FBase: A Replication Service for Data-Intensive Fog Applications

Jonathan Hasenburg, Martin Grambow, David Bermbach
TU Berlin & Einstein Center Digital Future
Mobile Cloud Computing Research Group
Berlin, Germany
{jh,mg,db}@mcc.tu-berlin.de

## ABSTRACT

The combination of edge and cloud in the fog computing paradigm enables a new breed of data-intensive applications. These applications, however, have to face a number of fog-specific challenges which developers have to repetitively address for every single application.

In this paper, we propose a replication service specifically tailored to the needs of data-intensive fog applications that aims to ease or eliminate challenges caused by the highly distributed and heterogeneous environment fog applications operate in. Furthermore, we present our prototypical proof-of-concept implementation FBase that we have made available as open source.

## KEYWORDS

Fog Computing, Data Management, Replication Service

**Reference Format:**
Jonathan Hasenburg, Martin Grambow, David Bermbach. 2020. FBase: A Replication Service for Data-Intensive Fog Applications. In *Technical Report MCC.2019.1.* TU Berlin & ECDF, MCC Research Group, 11 pages.

## 1 INTRODUCTION

Current state-of-the-art applications are typically deployed on top of cloud services; the cloud alone, however, is often not capable enough for emerging application domains such as autonomous driving, 5G mobile applications, eHealth, or the Internet of Things (IoT) [31]. Depending on the use case, reduced end-user latency, bandwidth limitations between sensors and cloud, or privacy challenges force developers to use fog computing instead: the combination of edge and cloud computing but also with optional small- to medium-sized data centers in the network between cloud and edge offers the best from both worlds [4, 28].

Due to these benefits, we have seen a number of fog applications emerge over the last few years, e.g., [7, 13, 18, 24]; however, one would still expect much more adoption of the fog computing paradigm. In [4], a number of possible reasons such as a lack of edge services or even hardware heterogeneity are discussed. Beyond these, we also see the problem of having to “reinvent the wheel”: developers need to start virtually from scratch for every fog application to *get data to where it is needed by a multi-tenant application in a highly distributed and heterogeneous environment.*

In this paper, we propose a replication service which provides a set of middleware abstractions specifically tailored for this task. These abstractions allow application developers to specify data placement and data movement using a declarative programming style while actual data distribution across multiple fog nodes is handled by the underlying replication service which we named FBase. In essence, our approach is novel in that FBase provides the abstractions and infrastructure for application-controlled replica placement. Therefore, we make the following contributions[1]:

- We identify a set of requirements for a replication service that aims to simplify the development of data-intensive fog applications (section 2).
- We propose and describe our replication service FBase and discuss how it addresses the identified requirements (section 3).
- We present our proof-of-concept prototype, which we have made available as open source, and its evaluation (section 4).

We also discuss related work (section 5) and our approach (section 6) before drawing a conclusion (section 7).

## 2 REQUIREMENTS

Data-intensive fog applications encounter a number of challenges; most of them are not new but they are significantly more pronounced than in existing cloud-based systems and, thus, require new solutions. The two most obvious challenges are the geo-distribution and heterogeneity of the runtime infrastructure. While cloud-based systems may run in a few geo-distributed data centers on top of more or less identical VM hardware, a fog-based system runs on a variety of machines. These can range from single board computers such as a Raspberry Pi to clusters of cloud VMs and anything in between, geo-distributed over at least hundreds of sites. For data-intensive applications, this means to handle replication and data distribution in such an environment.

Beyond these, resources at or near the edge are limited so that fog-based systems need to deal with much higher degrees of shared resources, not only at the level of infrastructure resources but also in the software stacks on top of that. Finally, fog-based systems need to interface with a variety of existing systems. These could be embedded cyber-physical systems at the edge, event brokers of all kinds, or stream processing systems and legacy applications in the cloud.

Ideally, a service as proposed in this paper deals with all these aspects to let application developers experience the same simplicity in the fog that they got used to while building cloud applications. We believe that this ideal situation is not achievable, e.g., complete distribution transparency is not feasible in the presence of faults.

[1]This is an extended version of [16].

A well-designed service, however, should provide suitable abstractions to handle the complexities of, e.g., geo-distribution, while not hiding the fact per se. Based on these premises, we identified the following main requirements for a replication service supporting data-intensive fog applications:

**Design for Multi-Tenancy:** Due to the higher degree of shared resources near the edge, it is not feasible to run several instances of the replication service (or alternative services) in parallel. Instead, such a service should be designed for multi-tenancy out of the box.

**Application-Controlled Data Placement:** Applications should be able to declaratively specify the placement of data. This means that they should control the placement of data while the underlying service handles data replication, movement, and distribution where and when necessary. This still exposes the fact that different sites exist but takes the hassle out of it.

**Hiding Infrastructure Heterogeneity:** An application should not have to worry about the number and kind of available machines at a particular site. Instead, these should be exposed through suitable abstractions, e.g., the total amount of available resources, while the service handles load balancing, scheduling, and resource management.

**Ability to Interface with Existing Systems:** A data-intensive fog application very likely needs to interact with other applications and systems, particularly at the edge but also in the cloud. A data-handling service should provide this functionality as part of the data management tasks as data-intensive applications will interact with other systems to either ingest or expose data. The application should be able to specify such interfaces in the same declarative way that it uses for data placement.

## 3 FBASE DESIGN

The main goal of FBase is to provide applications with the means to control data replication and data flow across geo-distributed sites using a declarative programming style. As a typical example, consider the following application: an IoT sensor produces data at the edge (e.g., a temperature sensor), triggers a local actuator (e.g., a smart blind), and buffers data at a nearby edge node. This data is then replicated to an intermediary node, e.g., at the regional level, where it is merged with data from other sensors, aggregated, and then replicated to the cloud. In the cloud, the aggregated historical data from a multitude of sensors is made available to web-based clients.

In this example, it is irrelevant to the application which physical machine handles the data at each site. As such, we developed the concept of nodes to *hide infrastructure heterogeneity*. In FBase, a node is a set of machines at a specific geographic location. Nodes self-organize and application clients (or other nodes for that matter) can choose to interact with any machine of a given node. We describe this concept in more detail in section 3.2.

In addition, the *application can control data placement* and data flows in a declarative way. For this, we propose the concept of keygroups which group data items that should be handled in the same way; each keygroup has metadata that describes which FBase nodes are handling the keygroup. FBase nodes can be involved in two roles (simultaneously): first, as replica nodes which persist data locally and serve application requests (e.g., get or put); second, as trigger nodes which passively listen to any updates on the keygroup data and expose these updates via an event-based interface. One purpose of the latter is to *interface with existing systems*; in our example above, this means triggering the local actuator at the edge and triggering the data merge on the intermediary node.

Node details, the geo-location of sites, and other information are available to applications through our cloud-based naming service. We describe the naming service in more detail in section 3.1 and the details of keygroup and data distribution in sections 3.3 and 3.4.

Finally, since keygroups are entirely isolated from each other, FBase is inherently *designed for multi-tenancy*. We describe more details on this in section 3.5.

Beyond addressing the requirements from section 2 as outlined above, we also describe the resulting consistency model of FBase in section 3.6 and how we imagine building on top of FBase in future work in section 3.7.

### 3.1 Naming and Configuration Management

FBase offers a number of tuning knobs through which applications can, for instance, control how data is replicated. Furthermore, for tasks such as access control or infrastructure management, it is necessary to assert that machines have unique IDs and that configuration data is stored with strict consistency guarantees. In contrast, application data can often tolerate eventual consistency [30].

For configuration data, we decided to follow a similar design as in GFS [12] and BigTable [6] which both use Chubby [5] to handle configuration data in a consistent way. Our "Chubby" is a component called *naming service* which handles naming (i.e., assignment of unique IDs) and storage of configuration data. IDs are immutable and are tombstoned when they are no longer needed. This means that other machines can safely cache configuration data and also share it with other machines so that the naming service itself is only involved when adding or removing machines. Overall, the naming service acts as the single point of truth for configuration data and naming in case of conflicts.

As the naming service stores no application data, which is likely to be updated frequently, its load is usually low; hence, it is unlikely to become a scalability bottleneck. Furthermore, it can easily be sharded to mitigate potential scalability problems.

We explicitly decided to introduce the naming service component instead of handling all naming and configuration management in a peer-to-peer (P2P) fashion for four reasons. First, having a consistent view on configuration data significantly simplifies the design and implementation of the rest of FBase. Second, especially when moving towards the edge, network bandwidth and hardware resources may be very limited and unreliable. Therefore, it is preferable to handle these tasks only on machines with sufficient resources. Having a dedicated component allows us to select these machines as part of the deployment process instead of having to reconfigure FBase instances to either run or not run this functionality. Third, a dedicated naming service can be scaled independently of the rest of the system as needed. Fourth, the naming service itself is only a component and does not make assumptions about its own implementation. In fact, our prototype comes with two implementations. Both are essentially wrappers that expose their internal strictly consistent storage structure as the naming service interface. One is based on the local file system and the other on Apache Zookeeper[2]; additional ones can be added easily.

Even though we believe that these reasons speak for our approach, in certain situations a “P2P naming service” can be a better option as, for example, configuration data cannot be updated when the naming service is unavailable. However, our remaining design aims to tolerate such situations temporarily, as explained in the following.

## 3.2 Nodes: Managing Collocated Machines

To hide the complexity caused by infrastructure heterogeneity and geo-distribution, FBase provides an abstraction for collocated machines called *nodes*, as such machines often have the same purpose and are used to scale-out systems. For instance, in an IoT scenario where sensor data is preprocessed at the edge before sending it to a cloud backend, a machine at the edge has to handle data of only a very limited number of sensors while the cloud backend has to handle the data from all edge devices. An obvious solution to this is to have several (preferably stateless) cloud machines behind a load balancer with a shared storage system. Therefore, nodes are groups of collocated machines at the same geographical site that have a shared purpose and use a shared data storage system for persistence. Overall, this means that FBase has two levels of infrastructure abstraction: on a node level, FBase is (more or less) a P2P[3] system of nodes (see also figure 1); within a node, FBase is a P2P system of machines with a shared persistence tier.

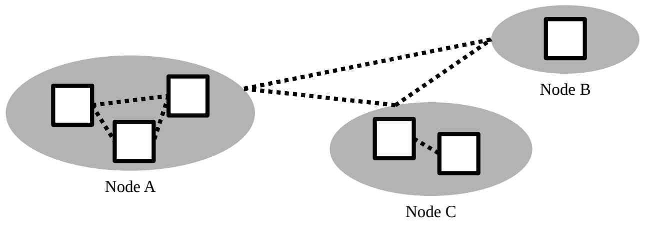

**Figure 1: FBase is a P2P System of Nodes which Comprise a P2P System of Machines Each**

This allows us to strictly separate data distribution functionality from scaling mechanisms: nodes have a unique name so that messages and data are always addressed to a node instead of to a specific machine. The machine of the target node that ends up processing this message is hidden from all entities outside of that target node[4]. The strict separation of responsibilities is also helpful for infrastructure membership management: at the node level, there will only be very low churn and even temporary unavailabilities of nodes can be expected to be infrequent due to the built-in redundancy. Also, this helps to keep the load on the naming service at a low level as the more frequent machine churn can be handled within nodes without involving the naming service.

[2]zookeeper.apache.org

[3]As the naming service handles some coordination logic, it is not a P2P system in the purest sense.

[4]Communication in FBase is done via pub/sub; nodes internally distribute responsibility for other nodes and the responsible machine subscribes to all updates from the sender node's machines.

In its simplest form, e.g., at the edge, a node is only a single machine running FBase and either an embedded database system or the local file system as a “shared” persistence tier. On the other end, e.g., in the cloud, a node may comprise a cluster of machines running FBase and either a dedicated shared storage system or a shared part (e.g., a database table) of a cloud storage service. In this design, we assume that the connectivity of nodes may be poor while connectivity within a node is mostly fast and stable. Figure 2 shows two example setups, one medium-sized node with a dedicated shared storage tier and one small node, e.g., a single Raspberry Pi, using the local file system for persistence. For production deployments, we generally recommend collocating instances of the storage system and FBase to facilitate low latency data access.

What happens at node level is based on configuration data stored in the naming service (see details in section 3.4). Within a node, however, machines must be able to self-manage, as they might not be able to connect to other nodes or some central server. This self-management comprises management of cluster membership, failover in case of machine crashes, load distribution, etc. We use the shared storage system for “communication” within a node so that machines can remain stateless. Furthermore, storing configuration data along with application data has the additional benefit that machines that can serve application requests also have access to the latest configuration data as well.

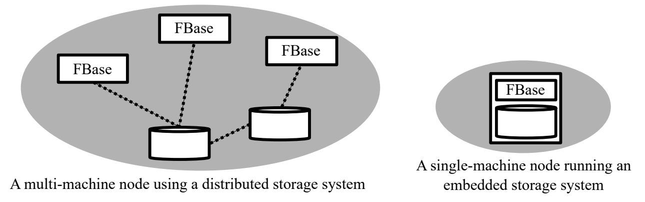
FBase
FBase
FBase
FBase
A multi-machine node using a distributed storage system
A single-machine node running an embedded storage system

**Figure 2: FBase Nodes can be Single Machines with Embedded Storage or Clusters of Machines with Shared Storage Systems**

## 3.3 Keygroups: Encapsulating Logically Coherent Data

Applications often have groups of data items that should be handled in the same way, i.e., they should use the same access policies, should be replicated in the same way, and will often be queried together. Typical examples for this are a sequence of time series values produced by a specific sensor or user records that would be stored in the same table of a relational database. Comparable to the entity groups in Megastore [1], FBase uses the concept of so-called *keygroups* for handling such groups of logically coherent data items which allows natural sharding.

Each keygroup has a globally unique name, some keygroup metadata, as well as the actual data records. The keygroup metadata holds a number of application-defined policies which specify how data should be replicated (see section 3.4) and who should have access to the data (see section 3.5). This is the key mechanism to give applications control over data distribution.

While keygroup creation and metadata updates (such as giving another party access) need to be confirmed by the naming service, each involved node also stores a copy to reduce load on the naming service and to reduce latency. This naming service dependence ensures that keygroups have globally unique names and that only authorized parties can access the respective data records.

The data records within a keygroup each comprise an unordered set of key-value pairs along with some per-record metadata, in particular update timestamps. As this abstraction is (on purpose) very similar to the BigTable [6] interface, it is relatively simple to utilize any of the widely used, scalable column stores as node persistence tier.

For multi-tenant setups, we propose to use a keygroup naming scheme that maps a keygroup to its tenant, e.g., by including the application id, user id, and data record description. This would lead to names such as "SmartHomeApp.SomeUser.Temperatures".

## 3.4 Distribution of Data

FBase provides two primary mechanisms for data distribution that are both specified on a per-keygroup level in the keygroup metadata: replication and transmission. For replication, a set of so-called *replica nodes* is defined that (i) each stores a copy of the respective keygroup's data records and (ii) accepts updates on data records of that keygroup which are then forwarded to all other nodes that are part of the keygroup's node set. Transmission, in contrast, is a mono-directional update propagation mechanism where the so-called *trigger nodes* receive updates from the replica nodes but have only read access to the data records. Trigger nodes specifically exist to integrate legacy applications or external systems such as a stream processing system through an event-based interface that exposes an update stream. As defined in section 2, developers should not have to worry about infrastructure management and heterogeneity. Thus, to replicate data to nodes in a specific geographic area, developers can query the naming service about adequate target nodes as the geo-location is part of the node metadata. Then, the only action left is to add these nodes to the keygroup; FBase takes care of the rest.

In practice, replica nodes may store incomplete replicas as applications can also specify a time-to-live (TTL) for each replica node of a keygroup, i.e., these replicas will store data records only for a limited period of time. This is particularly useful for edge nodes with limited storage capacity but can also be used to instantiate a buffered data stream that is accessible both as update stream and in an OLTP fashion. The combination of node types and TTL allows applications to specify arbitrary data distribution schemes as needed. See figure 3 for a typical IoT example use case: temperature information is ingested at the edge and buffered there for 10 minutes as defined by Keygroup 1. In addition, an external aggregator component based on a trigger node continuously reads the temperature information from the keygroup and forwards aggregated values to the cloud where they are stored persistently. Keygroup 2 ensures that all the aggregated data is replicated to a second cloud node. Note, that an FBase node can take the role of a replica node and trigger node of the same keygroup at the same time to fulfill both functionalities; in the example, a single node can act as Replica Node A and Trigger Node B.

For the actual distribution of data, FBase uses a publish-subscribe approach. This supports loose coupling of communicating nodes and, thus, improves overall availability as nodes simply publish

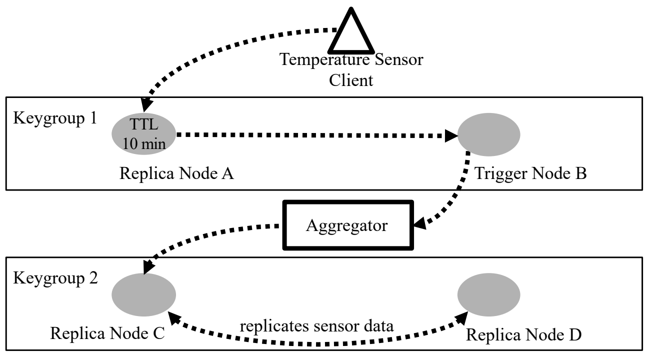

**Figure 3: Example: Using Replica and Trigger Nodes to Control Data Distribution**

updates in a fire-and-forget way similar to update propagation in PNUTS [10]. It is the responsibility of recipients to create the required subscriptions (based on the specifications made in the keygroup metadata). While this decouples individual nodes – which is most times the only feasible option for such geo-distributed deployments, it can lead to lost messages and out-of-order delivery. To mitigate these message delivery problems, messages use a counter as ID and nodes buffer the keys of data records for which they sent their most recent update messages. This way, recipients can detect missed updates and contact a sender node directly to request the latest version of the respective data record. In most cases, this ensures that messages are delivered reliably, and data becomes eventually consistent. However, a combination of machine failures, high update rates, and unfavorable buffer size and TTL settings may lead to permanently lost updates from the perspective of the receiving node. This is then the price to pay for high availability, low latency, scalability, and wide range of supported data distribution configurations which are more important for many applications than small amounts of lost data (e.g., temperature values from a sensor). Still, data loss can be avoided by leaving TTL disabled.

## 3.5 Security: Tenant Isolation and Access Control

While security is a complex topic, there is a variety of state of the art solutions that should be used. Furthermore, as FBase is about the programming abstractions for application developers, most questions of security are beyond the scope of this paper. We would like, however, to briefly outline how we realize tenant isolation and (to a certain degree) access control in FBase. For this, we need to protect both application data and configuration data from unauthorized access.

For the protection of application data, we assume that the network within a node is secure and that the machines involved are trustworthy. For instance, machines can add themselves to a node if they can access its storage system.

However, when transmitting data records between nodes or to application clients, these records are encrypted using AES128 with a keygroup-specific secret that is stored along with the keygroup meta-data. Instead of restricting access to data records, we only control access to this secret which is automatically updated whenever a node is removed from the keygroup's node set.

The communication necessary for the operation of FBase such as configuration changes is secured using RSA2048 (coupled with AES128), i.e., the data is encrypted with the recipients public key and signed so that the recipient can verify the identity of the sender. Furthermore, we added some basic authorization policies:

(1) When starting FBase, an initial node and an application client are created. Only existing nodes and application clients can authenticate themselves with the naming service, which is necessary for creating additional nodes and application clients.
(2) Nodes cannot add themselves to existing keygroups, but they can add other nodes to their own keygroups.

We believe that these measures provide sufficient security and tenant isolation under our assumed threat model. FBase can, of course, be complemented with other state-of-the-art security mechanisms and technologies.

## 3.6 Consistency Model

For all of the following explanations, it is important to understand that we divide data into two groups: configuration data and application data. Configuration data is stored with strict consistency guarantees as it describes the information required for the operation of FBase, e.g., information on cluster membership or keygroup secrets. Application data, however, is distributed based on asynchronous pub/sub communication and is stored in multiple, often only eventually consistent, data stores. Since nodes in geo-distributed fog networks are likely to fail and will often have high inter-node latency, we do not see a realistic alternative to asynchronous replication. Also, replication across eventually consistent data stores by itself would already result in eventual consistency [3]. Of the two consistency dimensions ordering and staleness, ordering is arguably the more challenging one for applications [2, 30], and we cannot avoid staleness. For ordering, however, we propose to follow the intuition behind the adaptive mastership in PNUTS [10] which is based on the observation that almost all data has a single writer, i.e., to size keygroups in a way that there will be only one concurrently writing client. For multi-writer keygroups, we plan to add an append-only keygroup type in the future which is inspired by the append-only logs of [31].

Beyond these, FBase does not provide any restrictions on application developers regarding the data distribution policies. As such, developers can design distribution schemes that result in frequent data loss, e.g., by setting TTL to zero. We do not want to restrict the configuration space in this regard as there might be applications that need precisely these configuration options, e.g., when only the latest data values are relevant.

## 3.7 Summary and Vision

FBase uses the concepts of nodes and keygroups to satisfy most of the requirements from section 2.

With nodes, FBase mostly hides the complexity of geo-distribution and infrastructure heterogeneity. From an application perspective, it is irrelevant how many machines a node comprises, each group of one or more machines at the same site is simply assigned a unique node ID and self-manages.

With keygroups, applications can control the placement of data to define arbitrarily complex replication schemes (including preprocessing in between) using a declarative programming style. In a similar fashion, applications can also integrate external systems such as legacy applications or stream processing systems. Furthermore, keygroups make it straightforward to use shared resources in multi-tenant setups, as each tenant can have its own namespace.

FBase is also a first step towards a comprehensive fog data management system. Specifically, we plan to work on predictive replica placement in which a prediction component will automatically adjust keygroup membership based on actual and anticipated physical client movement to ease the burden on applications while letting them retain control on replica placement.

## 4 EVALUATION

We evaluate FBase in three different ways: First, we give an overview of our proof-of-concept implementation that we created to show the feasibility of the design (section 4.1). Second, we present the results of several micro-benchmark experiments which we ran to evaluate the overheads created by FBase (section 4.2). Third, we describe how FBase could be used for the implementation of fog application scenarios (section 4.3).

## 4.1 Proof of Concept Implementation

To demonstrate the feasibility of our system design, we have implemented it as a proof-of-concept prototype in Java 8. Our prototype consists of three separate components – the FBase daemon, the naming service, and an application-side client library. In this section, we give an overview of our prototype which is also available as open source[5].

*FBase Daemon.* The FBase daemon is the software component that runs on every individual machine. Multiple machines running instances of the FBase daemon will organize themselves as a node if they are located at the same site and can connect to the same storage system. Each daemon serves application requests, publishes incoming updates to subscribers, and stores updates in the node storage system which makes the data available to all other deamons running on machines of the same node. The deamons also manage the subscriptions of the node, monitor the availability of other machines, and periodically verify whether the locally cached configuration information is still consistent with the state of the naming service. When a daemon realizes that it missed some updates from another node, e.g., when a machine within its node crashed or in case of network disruptions, it requests re-transmission of the corresponding data records.

Daemons map to the node concept in the following way: within a node, machines distribute responsibilities via the shared storage system. A machine that is responsible for handling keygroup "my.key.group" will subscribe to all machines of the other nodes in that keygroup for updates on the topic "my.key.group". The list of machines that are part of a node is currently stored by the naming

[5]https://github.com/OpenFogStack/FBase

service so that external machines will notice changes in membership (and thus potentially missing subscriptions) via the regular interaction with the naming service; in that case, missed update requests will be detected and requested again upon the next update. For future versions, we plan to only keep a set of seed machines listed in the naming service so that requests for node membership, i.e., other machines, are directed to the responsible node directly.

The FBase daemon uses an adapter architecture for connecting to storage systems; currently, it implements connectors for Amazon S3[6] (for cloud nodes) and in-memory storage (for single machine nodes). To add more connectors, e.g., a Cassandra[7] connector for small to medium sized nodes running in the fog, it is sufficient to implement our connector interface. All communication between FBase nodes as well as between nodes and the naming service is based on ZeroMQ [17] which offers additional message delivery guarantees beyond the ones implemented in FBase.

*Naming Service.* The naming service fulfills the tasks described in section 3.1; as mentioned, we currently have two implementations of it. The first uses Apache ZooKeeper for consistent and replicated storage of configuration data. In its current state, the implementation still requires some additional testing and bug fixing before it is ready to be used in practice. Therefore, we also provide a second naming service implementation that mocks the use of ZooKeeper based on the local file system – this could in fact be a viable distributed alternative when run on top of a distributed consistent file system. This implementation can be run as a single machine naming service and we used it in our experiment runs described in section 4.2.

*Application Client Library.* Applications can interact with FBase via a REST interface that is offered by every machine running the FBase daemon; we also implemented application-side client libraries in Java and Kotlin. These client libraries expose the FBase interface in a more accessible way for all JVM-based applications.

## 4.2 Micro-Benchmarks

In this section, we present the results of some micro-benchmarks with which we want to demonstrate that our prototype works properly and that the performance and staleness overheads created by FBase are appropriate for a research prototype.

*4.2.1 Latency Overheads.* With this experiment, we want to quantify the latency overhead for read and write operations, i.e., the latency difference between writing to a storage system directly or via FBase. For this purpose, we added debug level logging to FBase and deployed it on an Amazon EC2[8] m3.medium instance running the S3 connector[9]. On another m3.medium instance in the same availability zone, we installed a benchmarking client that sequentially issued 1000 put, read, and delete operations each to FBase. We measured both the end-to-end latency of FBase as well as the latency of writing from FBase to S3; table 1 shows our results.

**Table 1: Latency Measurements in ms**

| | Put | | Read | | Delete | |
|---|---|---|---|---|---|---|
| | FBase | S3 | FBase | S3 | FBase | S3 |
| Min | 50 | 11 | 10 | 6 | 40 | 6 |
| Max | 1275 | 1099 | 990 | 985 | 1292 | 1249 |
| Avg | 118 | 31 | 26 | 16 | 88 | 16 |
| Std Dev | 97 | 44 | 45 | 44 | 92 | 48 |
| $Q_{0.95}$ | 215 | 80 | 65 | 41 | 185 | 59 |
| $Q_{0.99}$ | 540 | 154 | 138 | 126 | 410 | 120 |

As one can see, the latency overhead of FBase (columns "FBase" minus "S3") for the read operations is rather small (10ms on average). In this time, FBase receives a request via HTTP from another machine, parses the request, and returns the requested data record. The overhead for write operations is larger, on average 87ms for put and 72ms for delete operations, since the updated data record also has to be provided to ZeroMQ which encrypts the data and publishes it asynchronously to other nodes (we publish data for this micro-benchmark even though there are no active subscribers deployed to evaluate the impact on the latency). We include the overhead of publishing in these measurements since an operation does only commit when a data update has been published to increase durability (if there is a network partitioning and FBase crashes after applying an update, ZeroMQ will still deliver the message upon reconnection unless the machine crashes.

Overall, these numbers show that the FBase approach is feasible for non-realtime use cases. It should be noted that our prototype has *not* been optimized for performance in any way as our goal was only to demonstrate the feasibility of the programming abstractions provided by FBase and to give a "proof of life" of our prototype.

*4.2.2 Staleness Overheads.* FBase distributes data asynchronously which means that replicas will incur staleness. In this micro-benchmark experiment, we aimed to quantify the staleness introduced by FBase alone (not including response times of storage systems or network latencies) as an additional measure of FBase' compute overhead. To achieve this, we deployed two FBase nodes and the benchmarking client on the same virtual machine; each running in their own Java VM. For this purpose, we used an m3.xlarge instance which offers four virtual CPU cores so that our nodes and benchmarking client would not compete for compute resources. We configured both FBase nodes to use the in-memory storage connector.

During the experiment run, the benchmarking client issued 1000 put and delete operations each as these two operation types can create temporary inconsistencies in FBase. For each operation of the benchmarking client, we measured the time between completing the write on the first and on the second node, i.e., the data-centric staleness [2]; see table 2 for our results. As one can see, the staleness overhead introduced by FBase itself is relatively low for both write operations. Compared to network latency in a geo-distributed deployment (which FBase is designed for), these are negligible.

[6]aws.amazon.com/s3
[7]https://cassandra.apache.org/
[8]aws.amazon.com/ec2
[9]Obviously, a fog deployment would use other storage systems but our purpose here is only to evaluate the overhead of FBase.

**Table 2: Staleness Measurements in ms**

| | Staleness FBase Put | Staleness FBase Delete |
|---|---|---|
| Min | 6 | 3 |
| Max | 98 | 57 |
| Avg | 15 | 8 |
| Std Dev | 11 | 9 |
| $Q_{0.95}$ | 36 | 28 |
| $Q_{0.99}$ | 60 | 53 |

## 4.3 Possible Usage Scenarios with FBase

The main purpose of FBase is to simplify the development of data-intensive fog applications. In this section, we describe three different usage scenarios; for each scenario, the implementation of an application would be difficult and labor intensive. Using FBase, however, only the application logic needs to be implemented, as all data management tasks are handled by FBase based on configuration details, i.e., in all three scenarios the data handling code requires only put and get requests as well as a few configuration instructions.[10]

*4.3.1 Remote Research Station.* Figure 4 illustrates a scenario with an offsite research station, e.g., in the polar regions. Here, recorded measurement information is buffered for one day at an edge node within the station and also replicated to a persistent cloud storage. A cloud-based processor retrieves the data from the cloud node, analyzes it, and makes the preprocessed data available to universities and research centers worldwide. For this, universities can evaluate the processed data on a local node which is automatically updated with all newly processed data. In the figure, there could be additional edge nodes on other research campuses that are also part of the replica node set of the keygroup on the right.

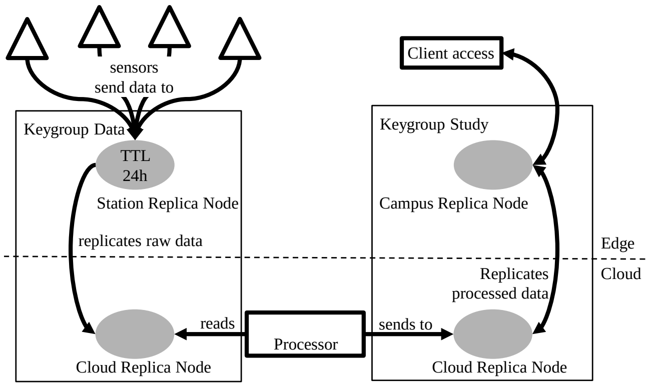

**Figure 4: Remote Research Station Scenario**

[10]We were originally planning to actually implement an example application but decided against it when we realized that almost all implementation effort would go towards implementing (completely unrelated) application-specific code. Instead, we decided to show a code listing for the complete data handling code of the most complex use case (the mobile app scenario).

*4.3.2 Carsharing Fleet Management.* In the scenario shown in figure 5, vehicle sensors collect information on the maintenance state of the car so that the carsharing provider can schedule predictive maintenance along with the corresponding fleet capacity planning. These sensors store their data inside the corresponding vehicle for 24 hours as defined by the related keygroup. In addition, a preprocessor application, also located in the vehicle, retrieves the stored data whenever a good communication uplink to the cloud is available. Then, it preprocesses the data, e.g., to reduce the size, and uploads the results to the Buffer Replica Node. The keygroup of the uploaded data specifies that the data should be transmitted to a trigger node which here acts as an event-based connector to the fleet management software. Similarly to figure 3, the same physical node can be both a replica node and trigger node, but it is also possible to use separate nodes. The setup can easily support additional vehicles with the preprocessors of these vehicles also sending the data to the same Buffer Replica Node.

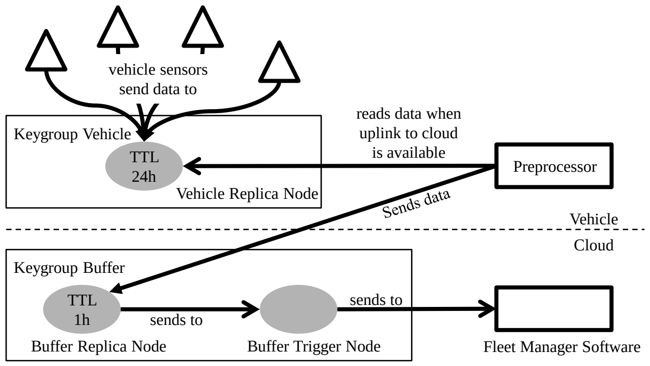

**Figure 5: Carsharing Fleet Management Scenario**

*4.3.3 Mobile App with Moving Client.* In the scenario shown in figure 6, a mobile device such as a smartphone or tablet runs an app that uses the cloud for persistence; to speed up data retrieval, data is also stored at the edge. Depending on movement of the user, the respectively closest edge node should be used. In the figure, keygroups are used to migrate data from one storage system – here placed at a cell tower – to another depending on the physical location of the client. In the example, the user moves from tower 1 to tower 2 which initiates the update to the keygroup configuration: the replica node of tower 1 is replaced with the node at tower 2. FBase can then either migrate the data between the two edge nodes or download it from the Cloud Replica Node to the Tower Replica Node 2 depending on latency and available bandwidth.

By just describing the desired results rather than required steps, it is effortless and straightforward to implement the replication and data management task with FBase. In fact, most keygroup-related tasks can be accomplished with a single line of code, e.g., adding clients and replica nodes[11]. Thus, the clients that operate in the fog environment can focus on their application-specific tasks. These, however, can still be quite complicated, so we refrained from actually implementing a full-fledged fog application for this evaluation;

[11]Trigger nodes and time to live configurations can be added to the keygroup in a similar fashion, i.e., by adding additional parameters to the method.

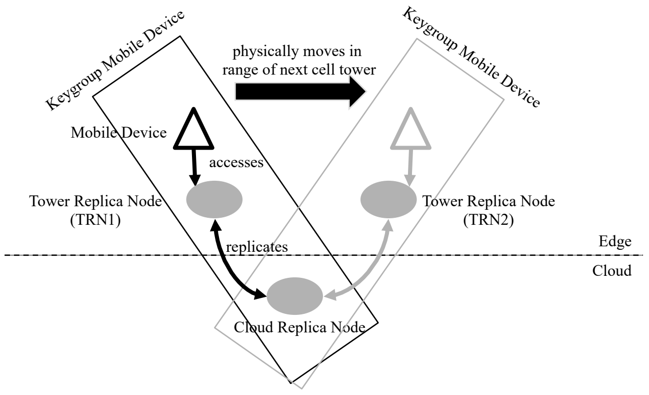

**Figure 6: Mobile App Scenario with Moving Client**

the code needed to interact with FBase would comprise only a few lines while most effort would have to go into implementing application-specific logic. See figure 7 which shows a Kotlin implementation of the FBase handling code using our Kotlin client library: the function SETUPKEYGROUP() should be invoked initially and (gets or) creates the described keygroup with two replica nodes. The function ONMOVEMENT() should be invoked whenever the mobile client connects to another cell tower. In a real application, parameters like the replica node IDs would probably be provided as function parameters.

```
fun setupKeygroup() {
    addToKeygroup(keygroupId = "mobile-app/mobile-device/data",
            clientIds = listOf("Mobile Device"),
            replicaNodeIds = listOf("TRN1", "Cloud Replica Node"))
}

fun onMovement() {
    removeFromKeygroup(keygroupId = "mobile-app/mobile-device/data",
            replicaNodeIds = listOf("TRN1"))

    addToKeygroup(keygroupId = "mobile-app/mobile-device/data",
            replicaNodeIds = listOf("TRN2"))
}
```

**Figure 7: Keygroup Setup and Client Movement Code**

## 5 RELATED WORK

Fog computing is still a relatively new computing paradigm. As such, there are many open research questions left, e.g., [4, 11, 27, 28], and there is only a limited number of existing publications in this research area.

Hourglass by Shneidman et al. [29] is closely related to FBase as it targets a similar problem. In Hourglass, so-called circuits describe the flow of IoT data from a data source to an end consumer. In contrast to FBase, Hourglass only supports buffering and storage if it is explicitly defined and not by default. Additionally, Hourglass does not consider replication of data items. Overall, the circuit idea focuses on the end-to-end data stream abstraction while FBase uses keygroups and their fine-grained data distribution configuration which enables arbitrary data replication and distribution schemes. This means that the basic unit in Hourglass is a single end-to-end stream where data is modified in between while FBase focuses on the individual stages where data is identical which can then be assembled into a larger data stream if required. As such, FBase provides much more flexibility and is better suited to the goal of supporting data-intensive fog applications. Furthermore, the focus on identical data sets instead of individual end-to-end streams avoids sending identical data over the same network connection several times while two circuits will always transfer the data twice.

With the Global Data Plane (GDP), Zhang et al. [31] propose a data management system design for IoT use cases. GDP is built around the notion of single writer logs with append-only semantics that are broken down into chunks. These chunks are placed inside a distributed hash table to support location-independent routing. GDP seems to be in a very early development stage and does neither state specifics on replica placement nor offer an implementation. In addition, it is bound to suffer from the downsides of hashing-based replica placement in the fog while FBase allows applications to freely specify arbitrary replication schemes as needed. The data log abstraction itself is a good fit for IoT use cases but not for other fog application domains for which FBase' keygroup abstraction offers more flexibility. In these applications, the GDP will quickly become overloaded as data can never be deleted so that it would benefit from a TTL feature as in FBase.

Mortazavi et al. [21] describe CloudPath, a platform that migrates application logic and data among nodes along the path between edge and cloud. These nodes can comprise one or multiple machines and use Cassandra for storage. In difference to the data replication of FBase, the CloudPath data replication is indirectly controlled by application functions which are also deployed on these nodes. CloudPath creates a local copy when the function needs access and removes the copy when the function has no further use of it. FBase, on the other hand, lets the application directly control replication which is especially useful when replicas should be created preemptively or for other purposes than low latency data access, e.g., to increase data durability. Furthermore, FBase supports applications running in any kind of execution environment while CloudPath applications have to run as platform-specific "PathExecute" containers.

In [23], Plebani et al. describe their vision of so-called virtual data containers to support the development of data-intensive fog applications. This approach seems to be in a vision stage and the focus is more on matching of application requirements to well-defined data sources controlled by other entities while FBase provides the building blocks for letting applications control data movement and replication of their own data.

With Nebula, Ryden at al. [26] propose a grid-inspired distributed edge store. Nebula centrally controls data placement in the so-called dataStore master. In a large-scale geo-distributed deployment, however, directing all requests first to such a centralized master server is prohibitive in terms of performance, scalability, and availability. In contrast, the naming service of FBase can be distributed and only (passively) handles metadata management rather than moving application data.

Based on their previous experiments [9], Confais et al. [8] try to make IPFS fog-ready. For this, they modify IPFS so that it does not read from the local file system but rather uses a local (per site) NAS system. While this solves part of the data locality problem identified in their previous work [9], IPFS is still based on the concept of immutable files which makes it more suitable for content delivery network use cases than for applications with frequent data updates: in such scenarios, IPFS will quickly hit the scalability ceiling.

To achieve lower latency for applications, Lin et. al [19] propose to locate a copy of the utilized database and an additional replication middleware also at edge nodes. While all transactions are executed locally, the middleware tracks the changes and propagates the "writesets" to the central "sequencer" which propagates updates to all other edge nodes. In contrast to FBase, this requires the sequencer as central coordinator which introduces a single point of failure. Moreover, the sequencer relies on stable connections to the edge nodes whereas FBase is specifically designed for unreliable environments. Finally, the sequencer uses full replication and does not allow arbitrary replication schemes as we do in FBase.

Psaras et al. [25] also identify the need for data storage at the edge to reduce the stress on networking resoureses. As a solution, they propose to use small data stores close to edge devices and discuss data management strategies which also include mobility aspects. However, their paper is more about the implications for the business models of service providers while we propose and evaluate a technical solution.

While the papers discussed above primarily deal with data management and migration in fog environments, a number of papers specifically aim to identify the "optimal" node for data placement. To solve this fog-related problem in geographically distributed systems with many instances, Gupta et al. [14, 15] as well as Mayer et al. [20] propose mechanisms to find suitable replication targets considering the physical data location and node stress levels. Naas et al. [22] formalize this assignment problem and propose a heuristic approach for solving it. For their approach, however, they require a lot of information on data characteristics and node capabilities which will often be hard to acquire, incomplete, and outdated in geo-distributed systems. Nevertheless, combining such approaches with FBase' keygroups might be a promising avenue to pursue.

## 6 DISCUSSION

FBase provides powerful abstractions to data-intensive fog applications: instead of handling data distribution themselves, such applications can simply provide configuration details to FBase which then manages all data accordingly. Using the abstractions of FBase, applications can create arbitrary complex data distribution setups with minimal efforts. In fact, we planned to let students participating in one of our teaching projects implement the usage scenarios presented in section 4.3 with the help of FBase, but had to learn that each scenario's implementation involves virtually no data handling code thanks to FBase. However, for very simple use cases such as connecting a single sensor to a backend service, application developers might be better off implementing data management themselves.

At this point, we would like to emphasize again that FBase is *not* a database system, FBase only offers the necessary infrastructure that allows developers to define replication paths, data flow, and fine-grained access control. However, this infrastructure could easily be used to implement a distributed fog datastore – in fact, we plan to do just that in future work: comparable to BigTable [6] offering added functionality on top of GFS [12], we imagine a fog datastore running on top of FBase that automatically handles replica placement (preferably through predictive replica placement), replica selection, and infrastructure management based on application-side monitoring.

For our initial FBase design, we decided to use a BigTable-like [6] data interface which is suitable for many but not all use cases. As such, FBase should not be used to handle large, unstructured, binary data. Likewise, applications that require more complex query functionality as provided by relational database systems or applications that primarily use timeseries data are not directly supported by FBase. However, the overall design of FBase could easily be adapted to provide also relational keygroups or append-only logs for timeseries data. Adding support for large unstructured binary data should be considered carefully: while it could easily be added as another keygroup abstraction, moving large data blobs around the fog may overload resources in practice; hence, the existing CRUD interface can be too coarse-grained for updates and data may need to be handled differently, i.e., not as a blob.

As already discussed in section 3.4, there are some unfavorable combinations of failure situations, TTL settings, replication configurations, and workload patterns which may not only lead to message loss but to actual data loss. As a simple example, a keygroup comprising an edge and a cloud node each may use the edge node only for buffering before transmitting the data to the cloud node which then is intended for persistent storage. With a TTL=60s setting for the edge node, any network connectivity problem between edge and cloud node that lasts longer than a minute will lead to data loss. FBase does not protect developers from making such a design decision – costs and benefits, particularly importance of data, should be carefully weighed. For instance, choosing a larger TTL value or using an additional keygroup only comprising the cloud node and explicitly deleting all received data in the shared keygroup might be a better alternative than the presented example in case of important data. FBase here provides several tuning knobs for the management of this tradeoff so that application developers can decide on a case by case basis how to handle it.

We have designed FBase for fog environments. As a "fog subset", FBase can also be used to manage data distribution in a cloud-only deployment. For example, it might manage data replication in a variety of cloud federation scenarios or it could be used to replace Amazon S3's cross-region replication with a more powerful alternative.

## 7 CONCLUSION

Emerging application domains such as autonomous driving or the Internet of Things often rely on edge computing, e.g., to circumvent bandwidth limitations or to profit from low latency communication with end users. However, as edge resources are inherently limited, fog computing as a paradigm which combines edge and cloud has recently emerged to combine the benefits of both worlds. While there are already a number of fog applications, we see – among others – the lack of supporting tools and services for running on and integrating both edge and cloud as one of the main reasons for the slow adoption of the fog computing paradigm.

In this paper, we identified a set of requirements that a replication service for data-intensive fog applications should satisfy. Based on these requirements, we proposed and described FBase, our replication service for data-intensive applications. FBase, for example, allows applications to simply describe how data should be distributed rather than handling data management themselves. As an evaluation, we presented our proof-of-concept prototype, the results of a number of micro-benchmark experiments, and three fog application scenarios that highly benefit from a service such as FBase.

## ACKNOWLEDGEMENTS

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 415899119.

## REFERENCES

[1] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proc. of CIDR.*

[2] David Bermbach. 2014. *Benchmarking Eventually Consistent Distributed Storage Systems.* Ph.D. Dissertation. Karlsruhe Institute of Technology.

[3] David Bermbach, Markus Klems, Stefan Tai, and Michael Menzel. 2011. Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs. In *2011 IEEE 4th International Conference on Cloud Computing.* IEEE, 452–459.

[4] D. Bermbach, F. Pallas, D. García Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai. 2017. A Research Perspective on Fog Computing. In *Proc. of ISYCC.* Springer.

[5] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proc. of OSDI.* USENIX.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of OSDI.* USENIX.

[7] Ning Chen, Yu Chen, Yang You, Haibin Ling, Pengpeng Liang, and Roger Zimmermann. 2016. Dynamic Urban Surveillance Video Stream Processing using Fog Computing. In *Proc. of BigMM.* IEEE.

[8] Bastien Confais, Adrien Lebre, and Benoit Parrein. 2017. An Object Store Service for a Fog/Edge Computing Infrastructure Based on IPFS and a Scale-Out NAS. In *Proc. of ICFEC.* IEEE.

[9] Bastien Confais, Adrien Lebre, and Benoît Parrein. 2017. Performance Analysis of Object Store Systems in a Fog and Edge Computing Infrastructure. In *TLDKS.* Springer.

[10] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. of VLDB* (2008).

[11] Manuel Díaz, Cristian Martín, and Bartolomé Rubio. 2016. State-of-the-art, Challenges, and Open Issues in the Integration of Internet of Things and Cloud Computing. *Journal of Network and Computer Applications* (2016).

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proc. of SOSP.* ACM.

[13] Martin Grambow, Jonathan Hasenburg, and David Bermbach. 2018. Public Video Surveillance: Using the Fog to Increase Privacy. In *Proc. of M4IoT.* ACM.

[14] Harshit Gupta and Umakishore Ramachandran. 2018. Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems.* ACM, 148–159.

[15] Harshit Gupta, Zhuangdi Xu, and Umakishore Ramachandran. 2018. Datafog: Towards a holistic data management platform for the iot age at the network edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18).*

[16] Jonathan Hasenburg, Martin Grambow, and David Bermbach. 2020. Towards A Replication Service for Data-Intensive Fog Applications. In *Proceedings of the 35th ACM Symposium on Applied Computing, Posters Track (SAC 2020).* ACM.

[17] Pieter Hintjens. 2013. *ZeroMQ: Messaging for Many Applications.* O'Reilly Media, Inc.

[18] Xueshi Hou, Yong Li, Min Chen, Di Wu, Depeng Jin, and Sheng Chen. 2016. Vehicular Fog Computing: a Viewpoint of Vehicles as the Infrastructures. *IEEE TVT* (2016).

[19] Yi Lin, Bettina Kemme, Marta Patino-Martinez, and Ricardo Jimenez-Peris. 2007. Enhancing Edge Computing with Database Replication. In *Proc. of SRDS.* IEEE.

[20] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. 2017. Fogstore: Toward a distributed data store for fog computing. In *2017 IEEE Fog World Congress (FWC).* IEEE, 1–6.

[21] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. 2017. Cloudpath: a multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing.* ACM, 20.

[22] Mohammed Islam Naas, Philippe Raipin Parvedy, Jalil Boukhobza, and Laurent Lemarchand. 2017. iFogStor: an IoT data placement strategy for fog infrastructure. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC).* IEEE, 97–104.

[23] Pierluigi Plebani, David Garcia-Perez, Maya Anderson, David Bermbach, Cinzia Cappiello, Ronen I Kat, Frank Pallas, Barbara Pernici, Stefan Tai, and Monica Vitali. 2017. Information logistics and fog computing: The DITAS approach. In *CEUR Workshop Proceedings.* CEUR-WS.

[24] Subhav Pradhan, Abhishek Dubey, Shweta Khare, Saideep Nannapaneni, Aniruddha Gokhale, Sankaran Mahadevan, Douglas C Schmidt, and Martin Lehofer. 2017. Chariot: Goal-driven Orchestration Middleware for Resilient IoT Systems. *ACM TCPS* (2017).

[25] Ioannis Psaras, Onur Ascigil, Sergi Rene, George Pavlou, Alex Afanasyev, and Lixia Zhang. 2018. Mobile data repositories at the edge. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18).*

[26] Mathew Ryden, Kwangsung Oh, Abhishek Chandra, and Jon Weissman. 2014. Nebula: Distributed Edge Cloud for Data Intensive Computing. In *Proc. of IC2E.* IEEE.

[27] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. 2016. Edge Computing: Vision and Challenges. *IEEE IoT-J* (2016).

[28] W. Shi and S. Dustdar. 2016. The Promise of Edge Computing. *Computer* (2016).

[29] Jeff Shneidman, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, Margo Seltzer, and Matt Welsh. 2004. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. In *Techn. Rep. TR-21-04 Harvard University.*

[30] Werner Vogels. 2008. Eventually Consistent. *ACM Queue* (2008).

[31] Ben Zhang, Nitesh Mor, John Kolb, Douglas S Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward A Lee, and John Kubiatowicz. 2015. The Cloud is Not Enough: Saving IoT from the Cloud.. In *HotStorage.* USENIX.