SessionStore: A Session-Aware Datastore for the Edge

Seyed Hossein Mortazavi*, Mohammad Salehe*, Bharath Balasubramanian †,
Eyal de Lara*, Shankaranarayanan PuzhavakathNarayanan†
*Department of Computer Science, University of Toronto
†AT&T Labs-Research

***Abstract*—It is common for storage systems designed to run on edge datacenters to avoid the high latencies associated with geo-distribution by relying on *eventually consistent* models to replicate data. Eventual consistency works well for many edge applications because as long as the client interacts with the same replica, the storage system can provide *session consistency*, a stronger consistency model that has two additional important properties: (i) *read-your-writes*, where subsequent reads by a client that has updated an object will return the updated value or a newer one; and, (ii) *monotonic reads*, where if a client has seen a particular value for an object, subsequent reads will return the same value or a newer one. While session consistency does not guarantee that different clients will perceive updates in the same order, it nevertheless presents each individual client with an intuitive view of the world that is consistent with the client's own actions. Unfortunately, these consistency guarantees break down when a client interacts with multiple replicas housed on different datacenters over time, either as a result of application partitioning, or client or code mobility.**

SessionStore is a datastore for fog/edge computing that ensures *session consistency* on a top of otherwise eventually consistent replicas. SessionStore enforces session consistency by grouping related data accesses into a *session*, and using a session-aware reconciliation algorithm to reconcile only the data that is relevant to the session when switching between replicas. This approach reduces data transfer and latency by up to 90% compared to full replica reconciliation.

I. Introduction

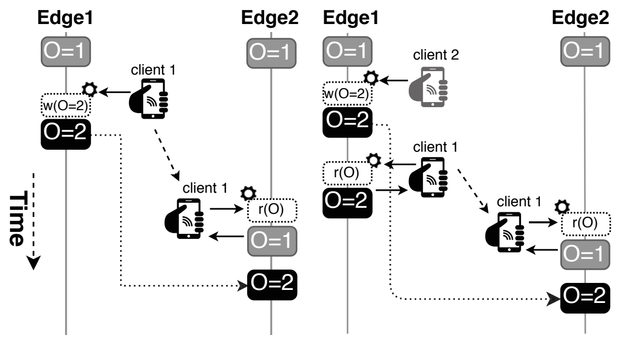
Edge computing expands the traditional cloud architecture with additional datacenter layers that provide computation and storage closer to the end user or device. For example, a wide-area cloud datacenter which serves a large country can be augmented by a hierarchy of datacenters that provide coverage at the city, neighborhood, and building levels. Edge computing facilitates next generation mobile and IoT (Internet of Things) applications that require low latency, or produce large data volumes that can overwhelm the network [1], [2].

Recent storage systems for the edge [3]–[6] generally rely on eventually consistent models [7], [8] to replicate data. These systems propagate updates in the background and guarantee that if no new updates are made to an object, eventually all replicas will converge to the same value. Eventual consistency works well for many applications where clients interact with the same replica for the duration of their sessions. The reason is that as long as the client interacts with the same replica, the storage system in effect provides *session consistency* [7], a stronger consistency model that has additional important properties: *read-your-writes*, where subsequent reads by a client that has updated an object will return the updated value or a newer one; and, *monotonic reads*, where if a client has seen a particular value for an object, subsequent reads will return the same value or a newer one. While session consistency does not guarantee that different clients will perceive updates in the same order, it nevertheless presents each individual client with an intuitive view of the world that is consistent with the client's own actions. Examples of applications that can benefit from session consistency on the edge include authentication services, file storage applications and messaging applications. We describe more usage scenarios for session consistency on the edge in Section III.

Session consistency however, may not be guaranteed when consecutive client requests are sent to different replicas. This may occur in edge applications when: (i) a mobile client switches between edges [9], [10]; (ii) functionality is dynamically reallocated between edges [11]; or (iii) an application's functionality has been partitioned between different datacenters [12]–[14] (e.g., running some functions on the edge and others on the cloud). If consecutive client requests are sent to different replicas before data needed by the client request is replicated, the application may not be able to read its own writes or have monotonic reads.

Figure 1 illustrates two such scenarios. In Figure 1a, *client 1* writes object O on Edge1. As a result of mobility, *client 1* switches its association to a different Edge2 and observes the old value of O on its subsequent read. In the second scenario illustrated in Figure 1b, *client 2* issues a command that results in object O being overwritten on Edge1. *Client 1* then reads this value and moves to Edge2. If *client 1* issues another request that reads object O on Edge2, an old value will be returned. While in the previous examples, clients read and write directly to the replicas of the storage system, this is done purely for ease of explanation. In practice, clients instead communicate with a replica of a service (e.g., an HTTP server) deployed on each edge datacenter that runs application code that access the replicated datastore.

We present SessionStore, a distributed datastore tailored for fog/edge computing that ensures *session consistency* between a hierarchy of otherwise eventually consistent replicas. Whereas previous approaches [15], [16] that provide session consistency on top of eventual consistent storage systems target applications running on a relative small number of cloud datacenters, SessionStore is designed for applications running on a large and variable number of edge/fog datacenters. SessionStore supports resource-limited data centres by leveraging partial replication and only replicating data on demand. SessionStore supports session consistency using a *session-aware* reconciliation algorithm that only reconciles keys that a client either



(a) Read your writes failure (b) Monotonic reads failure

Fig. 1: In (a) the client writes and Edge1 but its consequent read on Edge2 return an old value. In (2) client 1 reads a value but when it makes the read on Edge2, an old value is returned

reads or writes at the source replica. SessionStore further minimizes the data transfer by not transferring up-to-date data already existing on the destination. In our example application use case, this saving is as much as 95% in terms of data transfer.

The main contributions of this paper are: (i) a session-aware reconciliation algorithm that enforces session consistency by only transferring relevant client data; (ii) a prototype that implements our algorithms; (iii) an experimental evaluation based on micro-benchmarks and the RUBBoS benchmark that shows SessionStore is able to guarantee session consistency at a fraction of the latency and bandwidth costs of a strongly consistent implementation.

II. Related Work

Session consistency can in principle be provided by solutions that provide stronger consistency such as in [17] where all transactions are always executed at the local replica through snapshot isolation, or by geo-distributed transactional databases like Spanner [18], CockRoachDB [19] or by systems that use mixed consistency [20] and workload management [21] to provide strong consistency only where required. Unfortunately, these solutions use variants of distributed consensus that is very expensive across the wide-area-network [22], [23] and makes them impractical for the edge. As we show in our experiments in Section VI, enforcing strong consistency even for a moderate number of replicas incurs large latency costs.

A better approach is to use causally consistent systems like Bayou [24], COPS [25]. In practice, however, these approaches are also not applicable to edge deployments for three reasons: (i), these approaches assume a low and fixed number of replicas, whereas popular edge services may have hundreds or thousands of replicas. Many of these systems make use of vector time stamps where the overhead grows linearly with as the replication factor increases. While there are methods to trim the vector [26], [27], a compact vector clock that implicitly assigns vector positions to nodes requires centralized arbitration (or some other method of distributed consensus). Alternatively, the vector may include a unique node identifier like an IP address. In the latter case, however, significant additional storage is required, which makes using these systems on the edge unfeasible; (ii), these approaches assume full data replication (i.e., a complete copy of the database is stored at each site). Unfortunately, the resource-limited nature of edge datacenters dictates that they are only able to store a small fraction of the total state of a service or application. These limitations require the use of on-demand partial replication where only the state that is relevant to the current users of the edge datacenter is presently replicated on the edge datacenter; and (iii), data reconciliation is not fine-grained based on client or function data, rather reconciliation is done on table granularity. This approach results in high reconciliation latency and high bandwidth consumption for the transfer. This is particularly the case when only a fraction of the data is relevant to a given client.

Providing causal consistency on top of eventual consistency has been studied in [15] and [16]. In these studies a layer between the client and the data storage layer provides causal consistency for the client using vector clocks. In addition to the discussion on causally consistent systems presented above, we note that these works do not focus on session-aware reconciliation, which is crucial for our edge scenarios.

Other solutions to session consistency in the literature [7], [24], [28] use a combination of the following basic techniques: (1) sticky sessions can ensure that all reads and writes within session maintained by a client always communicates with a single replica, (2) maintain state at the client so that when a session does change replicas, the client can service requests from its cache till the new replica is up to date, and (3) use vector time stamps for the requests and ensure that each read or write is served or accepted at a replica in a manner as to satisfy the session guarantees. The first approach is not applicable to edge computing scenarios where the clients switch replicas over time due to client mobility or to access functionality deployed on different data centers. The second approach is only applicable when the client is fully trusted and has enough resources to store data. It is not practical for multi-user applications where raw data is kept at the server and is made available to clients in mediated form in response to explicit application requests. Finally, approaches that require vector time stamp break down when the number of replicas is large and dynamic as the overhead grows linearly with the replication factor. While there are methods to trim the vector [26], [27], they come at the cost of significant complexity and require strong coordination between replicas, which makes using these systems on the edge unfeasible.

In addition,various approaches have been proposed for application and service migration on the edge [11], [29], [30], however these approaches commonly depend on VM/Container migration methods or full application state synchronization. Our approach provides applications with flexible, fine grained data reconciliation through sessions. Our approach is specially advantageous for multi-user services that use the

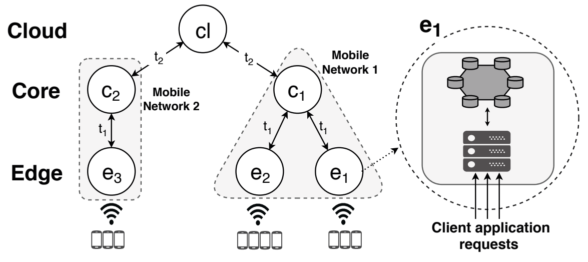


Fig. 2: Hierarchical datacenter topology

same replica to handle requests on behalf of multiple clients, where only a fraction of the replicated state is relevant to a given client.

There have been several works on data/state reconciliation in transactional databases [31], [32] and most key-value stores support some form of data reconciliation across replicas [33], [34]. However, in the former case, these solutions are heavy-weight, as necessary to guarantee ACID transactionality. The reconciliation function in common key-value stores are typically implemented as generic "stop-and-migrate" techniques that do not carefully track subsets of client data. This results in significant down times for the client. SessionStore, on the other hand, is much more light-weight since it guarantees session consistency as opposed to ACID transactionality.

This paper builds on workshop paper [35]. While this previous publication argued for the need for session consistency on edge datastores, it did not provide a complete design or a functional solution and included only a preliminary evaluation.

III. Use Cases

In this section, we give examples of scenarios that require session consistency to guarantee that requests emanating from the same client experience a view of the underlying data that is consistent with the client's own actions. We consider applications deployed on an edge network with two or more levels. For example, Figure 2 shows a sample edge network consisting of a cloud datacenter, and two mobile networks each with a datacenters at its core, and one or two additional datacenters at edge location such as base stations. Each datacenter has a replica of the datastore, as well as additional servers to run application code. We assume that the cloud datacenter stores a persistent full replica of the datastore. Each of the other datacenters hosts a partial replica and data gets replicated on-demand. On a read access, if the data is not already available on the local replica, it is fetched recursively from its parent. Similarly, updates are applied to the local replica and get propagated through the replica hierarchy in the background.

We enforce session consistency by grouping related datastore accesses into a *session* based on application-specific considerations. In the examples below, we uses a session to group together data accesses executed on behalf of the same user; however, it is possible to think of other applications where a session could be used to group together accesses executed on behalf of a device or a specific application module or function. We consider the case where requests that belong to the same session execute against different replicas due to: (i) user mobility; (ii) different parts of an application being deployed on different datacenters; or (iii) code mobility. The use cases below follow a stateless server design pattern where all application state is kept in the datastore, and applications are implemented as a collection of independent stateless functions.

Scenario 1: Mobile Client In this scenario, as a user moves around, his/her requests get routed to the closest edge datacenter. Session consistency is needed when the state that is read or written when connected to one edge datacenter is later accessed again after the user switches to a different edge. Consider the case of a user that leverages edge computing to edit a video. After recording a video on their phone, the user uploads it to an edge video-editing service which stores it in the datastore. The user then boards a bus, and proceeds to edit the video by applying a sequence of filters (e.g., image stabilization, cropping). By grouping the operations performed on behalf of the user into a single session, a datastore that provides session consistency guarantees that the effect of each of the filtering operations is preserved even as subsequent operation may run on different edges along the route as the bus travels.

Scenario 2: Functional Partitioning. In this scenario, an application's functionality is partitioned and deployed on different datacenters. Session consistency is needed when the results of executing one function on one datacenter should be made visible to another function running on a different datacenter. As an example, consider the case of a simple access control service that consists of three functions: *login*, *logout*, and *authorize*. A client logs into the system by providing a password to validate against a hash stored in the datastore. The *login* function is deployed on the *cloud* datacenter to ensure that sensitive password information is not replicated anywhere else. After successful validation, *login* adds a certificate with the user's permissions to the datastore. Similarly, to log a user out, *logout* modifies the certificate to indicate that it is no longer valid. Subsequent client requests (e.g., read an email, send a message) execute on one of the *edge* datacenters after first running *authorize*, which involves reading the user's certificate from the datastore to verify its validity. By grouping the operations performed on behalf of a client into a single session, a datastore that provides session consistency would guarantee that the version of the certificate created by the most recent invocation to *login* or *logout* is the one that is read by *authorize*.

Scenario 3: Function Mobility. In this scenario an application (or an application component) is reallocated between datacenters. Migration may be done for load balancing purposes, when the demands of a task surpasses the locally available resources on the current execution location, or to improve quality of experience. Session consistency is needed when after migration an application reads state from the datastore in the new datacenter that was either read or written in the old datacenter. As an example, consider the case of a interactive web hosted game that stores the state of the game in the datastore. When the network is experiencing low queuing delay, the application runs on the cloud, but migrates to a datacenter on the edge when an increase in wide-area traffic degrades the user's experience. By grouping the operations performed on behalf of each user into its own session, a datastore that provides session consistency guarantees that after migration the state of the game presented to the user corresponds to the user's last move.

The previous use cases can also run correctly on top of a datastore that provides stronger consistency guarantees, such as sequential consistency or casual consistency; however, the stronger properties come at a large cost in terms of bandwidth and latency as we show experimentally in Section VI. The above scenarios do not require a globally consistent view of the world; instead, they only require a view of the world that reflects the actions of operations that belong to the same session. The rest of this paper shows how session consistency can provide this guarantee with low overhead in terms of data transfer and replica switching cost.

IV. Design Considerations

In this section, we elaborate on our design choices for adding support for session consistency to a replicated datastore that runs on a hierarchy of data centers that facilitate edge computing. We consider three dimensions: when to synchronize state, what state to synchronize, and how to keep track or identify the state that needs to be synchronized.

Session consistency can be enforced either *proactively* or *reactively*. In a proactive implementation, data is continuously sent to other replicas eagerly. This approach supports fast switching between replicas; however, it results in high bandwidth consumption. On the other hand, a reactive implementation ensures session consistency only after a client switches to a new replica. Before running code on behalf of a client on a new replica, all relevant state has to be synchronized, which may incur delay.

We argue that the reactive approach is more appropriate for edge computing because the latency and resiliency demands of edge computing may dictate that mobile clients must often be served by their closest replica – thereby the proactive approach necessities a large number of service replicas and hence a very high replication factor that results in more resources needed on the edge. Our experiments confirm that a proactive implementation incurs large update latencies and high data volumes even for a modest replication factor. Conversely, the latency to switch between replicas that are kept in sync using reactive replication is relatively small (see Section VI-C3).

State between replicas can be synchronized using either *full replica reconciliation* or *session-aware data reconciliation*. In the former, the destination replica will have the latest/synchronized union of all records available at both replicas before the switch occurs. The advantage of this method is that it is conceptually simple, however it may result in high switching time and high bandwidth consumption for the transfer. In the latter, only data relevant to the session, including any records that were read or written, are synchronized. This approach is efficient in terms of data transfer and switching time; however, it is more complex and requires application support to identify relevant data accessed by the session. We argue that for multi-user services where the same replica handles requests from multiple clients, the second option where only the session's data is synchronized is more beneficial. Our experiments show that this approach reduces bandwidth requirements and latency. Moreover, in our experience the effort to label queries is modest.

To keep track or identify the state that needs to be synchronized, we can either tag individual records with read and write information, or use a higher level abstraction, such as user queries to capture access patterns. The benefit of tagging individual records is its simplicity, which comes at the expense of potential significant additional storage overhead for data object. Instead, we opted to track data accesses by recording SQL-like queries executed against the replica. While this approach is more complex to implement, it has lower storage requirements as simple queries can identify many data objects.

V. SessionStore

In this section we describe SessionStore, our distributed datastore for edge computing which guarantees *session consistency* on top of otherwise eventual consistent replicas. The basic idea behind our approach to ensuring *session consistency* is simple, yet effective: we group related datastore operations into sessions, and we track all the rows either read or written to by a session through tracking the queries it executes. When a client switches from a source to a destination replica, we ensure that the same (or newer) versions of the rows associated with their session are present on the destination replica before executing new queries.

In the rest of this section, we first describe a distributed datastore that provides eventual consistency across a hierarchy of replicas that extend from the cloud to the edge of the network. We next describe how we add support for session consistency on top of this otherwise eventual-consistent datastore.

A. Eventual-Consistent Operation

Our session consistent datastore is based on PathStore, an eventual-consistent object store introduced in [3]. PathStore is structured as a hierarchy of replicas configured as a tree with a persistent replica at the root, and an unlimited number of layers of partial replicas below it. Our implementation uses Cassandra [36] that can run on typical laptops [37] or even Raspberry Pi's [38], making it a feasible choice for edge deployments. Each replica runs a separate independent Cassandra ring, and our code is in charge of replicating data between otherwise independent rings. We replicate data at row granularity on demand in response to application queries. Each of the independent Cassandra rings may in turn consist of multiple servers and data may be internally replicated by Cassandra for fault tolerance or performance. In the rest of this paper, the term *replica* refers to a (potentially multi-server) Cassandra deployment on a datacenter in the PathStore hierarchy.

The datastore provides an API based on CQL, Cassandra's SQL dialect, which organizes data into tables, and provides atomic read and write operations at row granularity. CQL lets users read and write table rows using the familiar SQL operation `SELECT`, `INSERT`, `UPDATE`, and `DELETE`; however, CQL operations are limited to a single table – there is no support for joins.

Figure 3 illustrates a simple 3-level deployment of the datastore (a cloud replica, and two mobile networks each with a replica at its core, and one or two additional replicas at edge location such as base stations). To provide low-latency, all read and write operations are performed against the local replica. During a read query on a local replica, if the query has not been previously executed on the replica, we fetch it recursively from its parent. The query is then added in a *Query Cache* that keeps track of recently executed CQL queries. Subsequent CQL queries that match an existing entry in the cache are directly executed on the local node. Queries in the query cache are periodically executed in the background by a *pull daemon* to synchronize the local node's content with that of its parent (i.e., fetch new and updated records from the parent node). To reduce unnecessary processing, we keep track of the coverage of cache entries and the pull daemon bypasses queries that are otherwise subsumed by other queries that have a wider scope. For example the query `SELECT(*) FROM balloons` subsumes the query `SELECT (*) FROM balloons WHERE color=red`.

The datastore supports concurrent object reads and writes on all replicas of the hierarchy; updates are propagated up toward the root of the replica hierarchy in the background by a *push daemon*. Modifications are tagged with a version timestamp that records the time the row was inserted, and the ID of the replica where the modification was originally recorded. We assume that replicas are tightly synchronized using some accurate mechanism, such as GPS clocks. As modifications are propagated through the hierarchy (up by the *push daemon* and down by the *pull daemon*), we use the version timestamp to determine ordering – most recent timestamp wins.

Figure 3 illustrates the operation of PathStore for a simple table that keeps track of balloons of different colors and sizes. Initially (Figure 3a), the cloud replica stores 2 balloons, and all other replicas are empty. Figure 3b shows the result of running a query for small balloons (`SELECT(*) FROM balloons WHERE size=small`) on edge D: the small red balloon is first copied to the replica C and the query is added to edge C's query cache. From there, it is then pulled on to edge D. Figure 3c shows how the state changes after an application running on edge E adds two new balloons, one large green and one small blue. The push daemon of edge E propagates these two new balloons onto edge B. From there, the push daemon of B replicates the baloons onto the cloud replica. Figure 3d shows how the pull daemon on edge C identifies that there is a new balloon on the cloud replica that matches

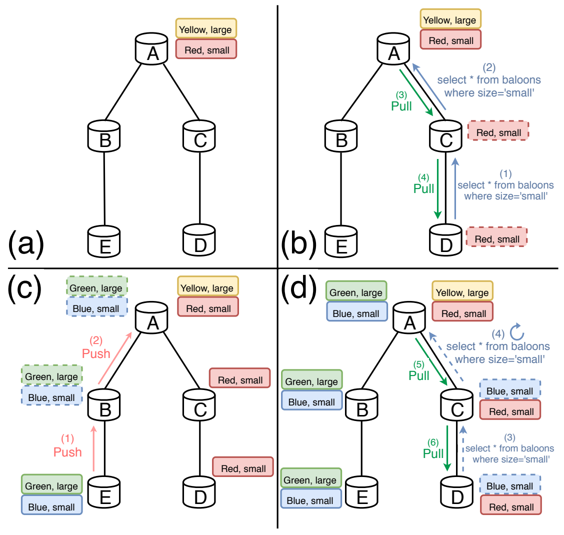


Fig. 3: PathStore operation.

the query in its query cache, and pulls the small blue balloon to edge C's replica. Similarly, the pull daemon on edge D also detects a new baloon on edge C that matches the query in its query cache and automatically pulls the small blue baloon onto node D.

B. Session-Consistent Operation

We next describe how we expand the above eventual-consistent implementation with support for session consistency across replicas. We fist describe how users can group database accesses into sessions. We then describe how we track data related to a session. Finally, we discuss how we perform session-aware replica reconciliation.

1) Sessions: We enforce session consistency by grouping related CQL requests into a *session*. What constitutes a session, however, is left to the application developer to determine. For example, the developer can decide to make a session representing a user, a device belonging to a user, a set of commands executed by a function, or a subset of the request issued by a device. Our system simply enforces session consistency semantics among those queries that are identified as belonging to the same session.

We identify each session using a Session Token, or *stoken*. The *stoken* consists of a four fields: A unique session id (*SID*), *timestamp*, *current replica*, and *status*. The *stoken* is encrypted and signed to prevent forging and misrepresentation. Developers chose between eventual and session consistency by including (or not) the *stoken* together with their queries.

2) State Tracking: To keep track of data related to a session, a *CommandCache* is added to each replica that stores all the queries that were executed on behalf of a session s.

For `INSERT`, `UPDATE` and `DELETE` commands, we keep track of modified rows affected by associated `SELECT` queries. For example if the session executes the command where $a1$ is the primary key (key):

$$\texttt{INSERT INTO } T_1(key,\ v1) \texttt{ VALUES } (a1, b1)$$

we store the following query in *CommandCache[s]*:

$$\texttt{SELECT * FROM } T_1 \texttt{ WHERE } key = a1$$

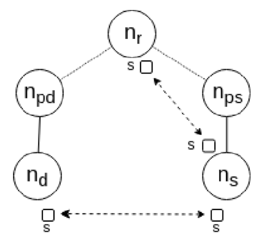
This transformation creates a query that tracks the accessed key $a1$.

The entries in the *CommandCache[s]* precisely identify the data accessed by a session. To recover the rows associated with a session we just have to execute the queries without any projections (`SELECT(*)`) and without any aggregations (without any `GROUP BY`). Our database implementation is based on Cassandra where queries are limited to a single table (no joins).

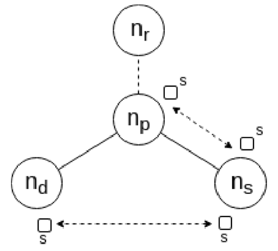
To keep the *CommandCache* small, we don't keep queries for a given session that are subsumed by more general ones. We also keep queries only for data that is actually replicated by each site. A background garbage collection mechanism deletes queries for sessions that have been moved to other datacenters.

To support session consistency, our current implementation can only run queries for a *stoken* at only one replica at a time. We keep track of the location of this replica on the *stoken* itself (the *current replica* field) and every site also keeps track of sessions it is serving.

3) Session-Aware Reconciliation: We leverage the *stoken* to detect when a client switches between replicas (e.g. when it moves between edge replicas n_s, n_r or n_s, n_d as shown in Figure 4a). When a replica receives a query it checks the *stoken*. If the ID of the replica servicing the query does not match the replica ID in the *stoken* then this is indicative that the client has switched replicas and the reconciliation process needs to start.



(a) non-siblings



(b) siblings

Fig. 4: Session transfer s between n_s, n_d

To assure session consistency, when a switching process is triggered on n_s, n_s's SessionStore replica will not process further commands for that session. Furthermore, requests for the session are delayed on n_d until the switch is complete. When the switching process is finished, it is reflected in the *status* field. If during the switching process the client moves to a another edge n_e, n_e will wait for the switching process on n_d to finish and then fetch the data from n_d.

During a switching process, n_d sends a request to the source replica asking for all rows modified or accessed by the session s. Having recorded all the queries executed by s, the source replica re-executes theses queries from its *CommandCache[s]*. It will then transmit the resulting rows as well as *CommandCache[s]* to the destination replica.

Viewer	Movie	Version	Rating
John	WALL-E	825968c0-195d-5d569c585662	10
Bob	Lion King	7adf7210-1958-59e16851d966	9
Susan	Bambi	6833c850-1958-59e16851d966	8
Anna	WALL-E	38400000-b23e-000044004725	10

TABLE I: Sample table $ratings$ on n_s

Viewer	Movie	Version	Rating
John	WALL-E	d33d7fe0-195f-5d569c585662	8
Mark	Cars	8b5f2471-19a2-59e168456212	9
Sara	Peter Pan	1263ca45-1912-59e36a58d990	8
Anna	Lion King	15460690-de22-a80b17057344	9

TABLE II: Sample table $ratings$ on n_d

Using queries to find accessed rows has the benefit of aggregation. While for writes we map every row modification to a separate query, for reads which usually dominates the accesses to the database, a single query can track many rows. Replication is done at full row level irrespective of columns projected in the select query.

Table I illustrates a database table *ratings* that keeps track of personalized movie ratings on n_s. Column *viewer* is the primary key and column *version* is added by SessionStore to determine ordering between updates to the same row. Now suppose that the following queries had been executed by session s on n_s:

```
SELECT * FROM Ratings WHERE viewer=ANNA

INSERT INTO Ratings(viewer, movie,
     rating) VALUES(Susan, Bambi, 10)
```

When the session switches from n_s to n_d, the two queries are executed on n_s (for the `INSERT` command, the associate `SELECT` query is executed) and only the third and fourth rows are copied to the *ratings* table on n_d as these are the only rows that match the recorded queries.

4) Optimizations: We implemented two optimizations to SessionStore's session-aware reconciliation that take advantage of data locality between different replicas and of SessionStore's hierarchical structure.

a) Δ-list optimization: The previous algorithm can be optimized when many clients are accessing the same rows on different replicas. The Δ-list optimization does not copy data that is already present at the destination replica. During a session switch, the destination, n_d, selects all primary keys and latest *version* for data belonging to an application and sends them to n_s. With this data n_s can then calculate rows accessed by a session that are either not already on n_d or have a newer version.

b) Sibling optimization: Finally, we provide a special *optimized-sibling-transfer* algorithm that works when the source and destination share a common parent node. This optimization takes advantage that in our design, all rows read by a node are also first replicated on its parent. In addition, the push daemon running on nodes, periodically propagates data written on a child onto the parent node. During a switch, only the rows that have not yet been pushed from n_s to n_p need to be replicated from n_s to n_d. Other rows can be accessed from n_p. Finally we synchronize any row on the destination that matches any query on the *CommandCache[s]* by fetching an update from the parent.

C. Failures

If a source replica fails when a destination is replicating state from it, SessionStore has to wait for the source to be available again and continue the transfer for the rows that it could not already replicate. The application is informed about any issue through an exception. The application can then decide to wait and retry, or invalidate the session and restart. Combining proactive replication to a few replicas with SessionStore's reactive approach is an avenue of future work.

VI. Experimental Evaluation

In this section we evaluate the performance of SessionStore and compare it to other alternatives for providing session consistency on a network of distributed replicas.

A. Platform

We conduct our experiments on an emulated hierarchical edge deployment shown in Figure 2. Our topology consists of a cloud datacenter, and two mobile networks each with a datacenters at its core, and one or two additional datacenters at edge location such as base stations.

Each (emulated) datacenter is implemented in a separate computer with $16GB$ of RAM and 8 CPU cores that runs either an instance of SessionStore, PathStore, or unmodified Cassandra, as well as an instance of Apache Tomcat. The network between the datacenters is emulated by using Linux's Traffic Control. Each link has a bandwidth of $1Gbps$. We assume that the underlay IP network has the same topology as the replica topology. This means that the point to point RTT between e_1, e_3 will be $t_1+t_2+t_2+t_1$. Unless stated otherwise, for the network latency between different datacenters, we optimistically assume two-way latencies $t_1 = 2ms, t_2 = 20ms$. These relatively low latency values tilts the comparison against SessionStore and in favor of Cassandra, which is more adversely affected by higher latency. Finally, requests are issued by clients running on additional computers that connect to one of the edge datacenters (e1, e2, e3) with negligible latency.

B. Workloads

Our evaluation uses a combination of locally-developed micro-benchmarks and RUBBoS [39], a benchmark application that models a discussion board. While RUBBoS was design as a web benchmark, we use it because its data access pattern is never the less representative of a typical multi-user application in three aspects: (i), it involves a large amount of state; (ii), it includes both read and write queries; and (iii), only a small fraction of the application's state is relevant to any given user.

The original RUBBoS benchmark is limited to text comments (1 KB in average), which are small compared to modern media-sharing standards. To better mimic the expected behaviour of a modern social media application, such as Snapchat or Instagram, which allow users to upload short videos and images, we create two new versions of the benchmark by adding an extra 10 KB or 100 KB of data to each comment to simulate a small and medium multimedia attachment. This increase the total size of the RUBBoS database from 540 MB to 23.9 GB and 240 GB, respectively. We used a RUBBoS database populated by over 2.34 million comments, 12000 stories, and 500000 users.

We used the Java Servlet-based RUBBoS implementation which was originally designed to store its state in relational database. We ported this code to use SessionStore instead. The ported benchmark uses eight tables and consists of roughly 40 different queries including `SELECT`, `INSERT`, `UPDATE` and `DELETE`.

C. Results

We next present results that quantify the overhead of keeping track of session information, the benefits of session-aware reconciliation, compare the approach to alternatives that enforce stronger consistency as the cost of higher overhead, and explore the sensitivity of SessionStore session-aware reconciliation protocols to the number of queries in the command cache.

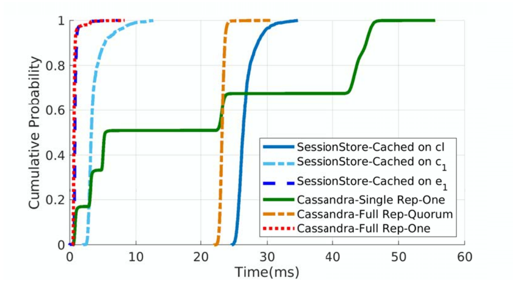
1) Session Tracking Overhead: To measure the cost of keeping track of session state, we compared the latency for reading and writing single $1KB$ row on e_1 with SessionStore. The experiment is repeated for 10000 different rows. Figure 5a shows a CDF of the read latencies for SessionStore in three different scenarios that assume the rows being read are already replicated on e_1, c_1, and c_l, respectively. The read latency for SessionStore is indistinguishable from PathStore(not shown), which indicates that the session tracking overhead is negligible. As expected, the figure shows that replication at the edge reduces read times dramatically. The average time to read a row already available on the edge was 0.9 ms, compared to an average of 4.65 ms and 26.2 when the row had to be fetched from the core and cloud, respectively.

Figure 5b shows a CDF of the write latency for SessionStore. There is only one configuration as all writes are preformed on the local replica (e_1). The average write time is 0.73 ms, and is similarly indistinguishable from write time in PathStore(not shown).

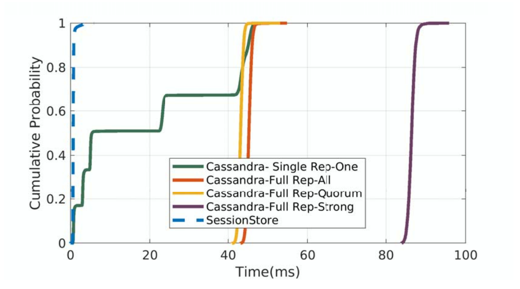
2) Session Migration: We use the RUBBoS benchmark to evaluate the costs in terms of latency and bandwidth of enforcing session consistency when a user switches between two replicas as a result of mobility. We consider four different approaches: Full-replica reconciliation, session-aware reconciliation, $\Delta-$list optimization, sibling optimization.

		Full Reconciliation (e_1, e_3)	Session-Aware (e_1, e_3)	Δ-List (e_1, e_3)	Neighbor (e_1, e_2) No users on e_2	Neighbor (e_1, e_2) 100 users on e_2
Default RUBBoS rows	Data Transfer	1.86 MB (40 KB)	187.25 KB (73.3 KB)	141.22 KB (45 KB)	14.2 KB (1.2 KB)	198 KB (38 KB)
	Time	562.1 ms (20 ms)	343.9 ms (57.8 ms)	288.52 ms (45 ms)	15.9 ms (1.4 ms)	62.7 ms (17.9 ms)
Added 10KB to each row	Data Transfer	22.7 MB (0.10 MB)	2.56 MB (563.1 KB)	1.22 MB (220 KB)	16.3 KB (2.2 KB)	1.16 MB (70 KB)
	Time	2.24 s (32 ms)	534.0 ms (40.59 ms)	330.4 ms (30.1 ms)	19.2 ms (2.4 ms)	76.86 ms (12.48 ms)
Added 100KB to each row	Data Transfer	220.3 MB (1.1 MB)	24.32 MB (6.6 MB)	15.9 MB (4.2 MB)	13.1 KB (4.3 KB)	10.7 MB (1.7 MB)
	Time	10.7 s (76 ms)	1.09 s (169.1 ms)	741.51 ms (105.1 ms)	20.8 ms (3.7 ms)	153 ms (25.5 ms)

TABLE III: Average reconciliation time and data transfer for a Rubbos client. Standard deviation in parenthesis.



(a) Reads



(b) Writes

Fig. 5: CDF of latency required to read and write a $1KB$ row.

We use the client emulator in the RUBBoS package to simulate 100 clients connected to replica on e_1 that are browsing and commenting on the RUBBoS bulletin board. The client emulator sent HTTP requests to Servlets running on the edge node e_1 which generated 2203 queries on the SessionStore replica on e_1. This resulted in SessionStore fetching data from cl and replicating it on e_1. A total of 1.86 MB for the text-only version of RUBBoS, and 22.7 MB and 220.3 MB for the versions with the small and medium multimedia attachments was transferred. On average, each RUBBoS query resulted in 13.4 rows on the database that exemplifies the benefits of using a query based approach compared to tagging each row.

Table III shows the latency and data transferred for different replica reconciliation scenarios for a client that moves from e_1 to either e_2 or e_3. The experiment is repeated 100 times, once for every client. We first consider the worst case where sessions move from e_1 into a cold e_3 replica that does not have any data. Full-replica reconciliation (first column) requires sending the full $1.86, 22.7, 220.3$ MBs of application data which takes 562.1 ms, 2.24 s and 10.7 s for each of the three configurations of the benchmark. In contrast, session-aware reconciliation (second column) only transfers an average of $0.18, 2.56, 24.32$ MBs of data, which takes only $343.6, 534, 1090$ ms. This major improvement, represents a reduction in data and latency of close to 90%, and is strong evidence of the benefits of leveraging session-aware reconciliation for server applications where only a fraction of the replicated data is relevant to a given client.

The Δ-list optimization (third column) further improves these numbers. In this experiment, we assume that a different set of 100 clients send requests to e_3 before the transfer. e_1 calculates the rows it needs to send to e_3 for each user and on average transfers 141.2KB's of data. For the three version of RUBBoS, Δ-list optimization only transfers $0.14, 1.22, 15.9$ MB of data in $288, 330, 741$ ms, which is an additional $22-35$ percent improvement in each scenario compared to the Session-Aware approach. Δ-list performs best when each row contains a lot of data and saves on bandwidth and transfer time by not sending those rows that are already on the destination.

We next evaluate the benefits of the sibling optimization when a single client moves from e_1 to e_2. We first consider the case where there are no other users on e_2 (fourth column). This results in only information about the queries transferred between the two nodes which is only 16.5 KB of data and takes less than 20 ms on average for the transfer (compared with 10.7s with the Full Reconciliation approach) . This is extremely fast compared to other scenarios because no other data needs to be transferred between the nodes. If the user executes their commands again, the data will be fetched from c_1 so the cost of fetching the data will be on demand and when the user requires it. Finally, we assume a scenario where another set of 100 users run the same application (and hence run similar queries) on e_2. Common queries between the moving user and users already running on e_2 may result in synchronizing data that from the parent. This on average increases the transfer time to $62, 76, 153$ ms and a further $0.19, 1.16, 10.7$ MB of data is transferred between e_2, c_1. In this particular application, many queries are common between sessions so more stale data has to be fetched from the parent.

3) Comparisons with Eager Replication and Strong Consistency: In this section, we explore three alternative ways in which *unmodified* Cassandra could be deployed on our

	Scenario	Average data transfer per row
Reads	SessionStore Fetch from cl	3245.8B
	SessionStore Fetch from c_1	1620.7 B
	SessionStore Fetch From e_1	0
	Cassandra Full Replication	0
	Cassandra Single Replication	1120.7 B
Writes	SessionStore	2346.8 B
	Cassandra Full Replication	6372.4 B
	Cassandra Single Replication	1213.6 B

TABLE IV: Data transfer

network of six datacenters that use eager replication or strong consistency to guarantee session consistency for a client that can move between replicas. In these configurations, all six datacenters form a single Cassandra ring, and each Cassandra server creates point to point connections to other servers using the underlying IP network.

Full Replication-All, uses a replication factor of six and Cassandra's *All* consistency model which requires all replicas to respond before a write operation returns. We can then use Cassandra's *One* consistency model for the reads which will fetch data from the local replica. *Full Replication-Quorum*, also uses a replication factor of six and Cassandras *Quorum* consistency model. This configuration requires a responses from a quorum of replicas for both reads and writes. *Single Replication-One*, uses a replication factor of one, and relies on Cassandra's standard hashing algorithm to uniformly distribute rows among replicas in the Cassandra ring. Reads and writes in this configuration involve a single server. Finally to compare to a strongly consistent database, *Full Replication-Strong* acts similarly to Full Replication-All with addition of linearizable consistency through the use of Cassandra's light weight transactions and the Paxos protocol [40].

Figure 5 shows the CDFs of the time required for writing or reading a single $1KB$ row on e_1. The experiment is repeated for 10000 different rows. Table IV shows the average data transferred aggregated across all links to store or read a $1KB$ row for the various configurations. All Cassandra alternatives perform poorly, which is hardly surprising given that Cassandra is not designed to be used in this manner and requires communication between different servers. On the other hand, our results are optimistic as real-world edge deployments will likely consist of a much larger number of data-centers.

Full Replication-All handles reads very well, but pays for it with high latency and bandwidth cost for writes. *Full Replication-Strong* performs even worse as the Paxos protocol needs additional rounds of communication between nodes. *Full Replication-Quorum* is a little better for writes, but much worse for reads. Finally, *Single Replication-One* read and write performance varies widely between rows based on their random allocation across the various data-centers. In comparison, SessionStore provides low latency for writes and reads, particularly in cases where the row are already available on e_1 or c_1, and uses much less bandwidth.

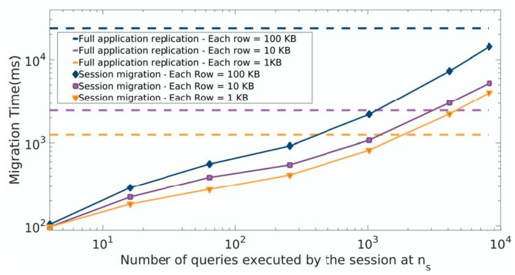


Fig. 6: Comparing session reconciliation(solid lines) and full application data reconciliation that consists of 10000 rows (dashed lines). Both axes are in logarithmic scale

4) Size of Command Cache: We evaluate the benefits of session-aware reconciliation as a function of the fraction of data in the replica that is relevant to a session and the number of queries used to track this data.

Figure 6 plots the latency to reconcile 10000 rows when a session moves from $n_s = e_1$ to $n_d = e_3$. We consider two reconciliation strategies: Full reconciliation, depicted by the dashed lines, that does not keep track of data accessed by individual sessions, and as a result all 10000 application rows have to be copied when the client moves between replicas. This becomes specially expensive when the amount of data stored in each row increases ($1KB, 10KB, 100KB$). Session-aware, displayed as solid lines, uses the *CommandCache* to keep track of rows accessed by the client that need to be moved between the replicas. We vary the number of commands executed by the client between $1, 8192$ and we assume each command only effects a single row. When the mobile client accesses only a fraction of the total data used by the service it is more beneficial to track session data. However, as the number of queries for a session increases, the overhead also increase because each query in the *CommandCache* has to be fetched and executed. As expected, the benefits of session-aware reconciliation is more distinguishable as the as the amount of data in each row increases. As shown in the Figure, when the rows are $1KB$, after around 1200 commands executed at n_s, it takes less time to transfer the full application data (orange lines). But when each row contains $100KB$s, even by executing 8192 commands for the session at n_s, it is still faster to use session-aware reconciliation (blue lines).

VII. Conclusions

A key tenet of fog computing is the ability for clients and application functions to be redirected seamlessly across the different edge data centers hosting the data replicas of a service or application. In this paper, we present SessionStore, a novel storage system that provides session consistency even when the client switches between replicas in different edge locations. Our session-aware reconciliation algorithms enforces session consistency at minimal costs, by tracking the accessed or effected keys by a session and then performing fine-grain reconciliation on the destination replica with minimum overhead. Our results show that our approach provides session consistency at a fraction of the latency and bandwidth costs of a system with eager replication or strong consistency, with minimal transfer costs. As future work, while our reconciliation algorithms ensure that only data pertaining to a session is migrated, we wish to explore other optimizations that reduce the data transfer.

References

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[2] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: a complete survey," *Journal of Systems Architecture*, 2019.

[3] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara, "Cloudpath: a multi-tier cloud computing framework," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, p. 20.

[4] B. Confais, A. Lebre, and B. Parrein, "Performance analysis of object store systems in a fog and edge computing infrastructure," in *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXIII*. Springer, 2017, pp. 40–79.

[5] I. Psaras, O. Ascigil, S. Rene, G. Pavlou, A. Afanasyev, and L. Zhang, "Mobile data repositories at the edge," in {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[6] H. Gupta, Z. Xu, and U. Ramachandran, "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)*, 2018.

[7] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[8] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, no. 2, pp. 37–42, 2012.

[9] L. F. Bittencourt, J. Diaz-Montes, R. Buyya, O. F. Rana, and M. Parashar, "Mobility-aware application scheduling in fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 26–35, 2017.

[10] J. Plachy, Z. Becvar, and E. C. Strinati, "Dynamic resource allocation exploiting mobility prediction in mobile edge computing," in *2016 IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*. IEEE, 2016, pp. 1–6.

[11] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

[12] G. Bianchi, E. Biton, N. Blefari-Melazzi, I. Borges, L. Chiaraviglio, P. de la Cruz Ramos, P. Eardley, F. Fontes, M. J. McGrath, L. Natarianni *et al.*, "Superfluidity: a flexible functional architecture for 5g networks," *Transactions on Emerging Telecommunications Technologies*, vol. 27, no. 9, pp. 1178–1186, 2016.

[13] A. Tiwari, B. Ramprasad, S. H. Mortazavi, M. Gabel, and E. d. Lara, "Reconfigurable streaming for the mobile edge," in *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*. ACM, 2019, pp. 153–158.

[14] Q. Zhang, Q. Zhang, W. Shi, and H. Zhong, "Firework: Data processing and sharing for hybrid cloud-edge analytics," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 9, pp. 2004–2017, 2018.

[15] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai, "A middleware guaranteeing client-centric consistency on top of eventually consistent datastores," in *2013 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2013, pp. 114–123.

[16] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 761–772.

[17] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris, "Enhancing edge computing with database replication," in *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 2007, pp. 45–54.

[18] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[19] "Cockroachdb: Ultra-resilient sql for global business." [Online]. Available: https://www.cockroachlabs.com

[20] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Presented as part of the 10th* {*USENIX*} *Symposium on Operating Systems Design and Implementation* ({*OSDI*} *12)*, 2012, pp. 265–278.

[21] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 292–308.

[22] A. Ailijiang, A. Charapko, M. Demirbas, B. O. Turkkan, and T. Kosar, "Efficient distributed coordination at wan-scale," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 1575–1585.

[23] K. Lev-Ari, E. Bortnikov, I. Keidar, and A. Shraer, "Modular composition of coordination services," in *USENIX Annual Technical Conference (ATC)*, 2016.

[24] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. IEEE, 1994, pp. 140–149.

[25] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 401–416.

[26] N. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves, "Dotted version vectors: Logical clocks for optimistic replication," *arXiv preprint arXiv:1011.5808*, 2010.

[27] R. Prakash and M. Singhal, "Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems," *Wireless Networks*, vol. 3, no. 5, pp. 349–360, 1997.

[28] "Configure sticky sessions for your classic load balancer," 2018. [Online]. Available: https://docs.aws.amazon.com/elasticloadbalancing/\latest/classic/elb-sticky-sessions.htm

[29] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K.-K. R. Choo, and M. Dlodlo, "From cloud to fog computing: A review and a conceptual live vm migration framework," *IEEE Access*, vol. 5, pp. 8284–8300, 2017.

[30] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile edge clouds," *IEEE Wireless Communications*, vol. 25, no. 1, pp. 140–147, 2018.

[31] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 494–505, 2011.

[32] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 301–312.

[33] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[34] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.

[35] S. H. Mortazavi, B. Balasubramanian, E. de Lara, and S. P. Narayanan, "Toward session consistency for the edge," in *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[36] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[37] "Cassandra hardware choices," December 2017. [Online]. Available: http://cassandra.apache.org/doc/latest/operating/hardware.html

[38] "Installing datastax enterprise on raspberry pi 2 with ubuntu core os," December 2017. [Online]. Available: https://medium.com/@johnsercel/datastax-cassandra-raspberry-pi-2-b6547eb43217

[39] C. Amza, E. Cecchet, A. Chanda, A. L. Cox, S. Elnikety, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Specification and implementation of dynamic web site benchmarks," in *5th Workshop on Workload Characterization*, no. CONF, 2002.

[40] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, pp. 18–25, 2001.