Towards Session Consistency for the Edge

Seyed Hossein Mortazavi[1], Bharath Balasubramanian[2], Eyal de Lara[1], and Shankaranarayanan Puzhavakath Narayanan[2]

[1]Department of Computer Science, University of Toronto
[2]AT&T Labs-Research

Abstract

We describe a distributed datastore tailored for edge computing that provides session consistency between otherwise eventual consistent replicas. Existing solutions for session consistency assume that the client is associated with the same replica through a session. However, in edge computing use-cases, a client interacts with multiple replicas housed on different datacenters over time, either as a result of application partitioning, or client mobility. Our core algorithmic innovation is our client reconciliation algorithm that enforces session consistency by tracking and migrating only the client-affected keys between the replicas. Our results show that our approach provides session consistency at a fraction of the latency and bandwidth costs of a strongly consistent system, and with reasonable migration costs.

1 Introduction

Edge computing expands the traditional cloud architecture with additional datacenter layers that provide computation and storage closer to the end user [14, 5, 6, 8]. For example, a wide-area cloud datacenter which serves a large country can be augmented by a hierarchy of datacenters that provides coverage at the city, neighborhood and building level. By adding datacenters closer to the client device, edge computing makes possible next generation mobile and IoT applications that require low latency or that produce large volumes of data [16, 9, 13].

Edge computing provides an opportunity to re-envision the way web services are deployed. Whereas existing web applications and services may be *replicated* across wide-area cloud datacenters to improve scalability and fault tolerance, edge computing encourages *partitioning* service functionality by placing components or functions at the datacenter layer that best meets performance and security requirements. For example, a wearable smart assistant could execute latency sensitive and bandwidth intensive functions, such as face recognition on a nearby edge datacenter, while running infrequent and latency tolerant functions, such as user authentication and preference editing, on a traditional wide-area cloud datacenter.

Web services are typically deployed on top of storage systems that relax consistency between replicas in order to provide high availability and performance [1]. A common approach is to use *eventual consistency*, where the storage system guarantees that if no new updates are made to an object, eventually all reads will return the last updated value [10]. Purely eventual consistent systems are very hard to reason about. Instead, web applications typically assume a stronger model called *session consistency* that provides two additional useful properties: *read-your-writes*, where subsequent reads by a client that has updated an item will return the updated value or a newer one; and, *monotonic reads*, where if a client has seen a particular value for the object, subsequent reads will return the same value or a newer one [18].

Session consistency is easy to implement by the use of *sticky sessions* [1, 19, 2, 3], where all reads and writes within a session maintained by a client are directed to the same replica. Sticky sessions work very well for applications deployed on the wide area cloud, where (in the absence of failure) a client communicates with a single cloud datacenter for the duration of the session. Unfortunately, this is not the case for edge computing where a client interacts with multiple replicas housed on different datacenters over time, either as a result of application partitioning, or client mobility and data/state migration for edge computing remains a challenge [15]. Existing solutions such as Spanner [7], which provides global consistency between geo-distributed replicas, or Bayou [17], which provides causal consistency between weakly-connected or disconnected replicas do not scale for edge deployments because, while these approaches assume a low replication factor, popular edge services may have hundreds or thousands of replicas.

In this paper, we argue that a storage system designed for web service deployment on edge datacenters should include support for session consistency across multiple replicas located on different datacenters. We describe the implementation of such a system based on PathStore [12], a hierarchical wide-column store that (previous to this paper had only) support for eventual consistency. A PathStore hierarchy consists of a persistent replica at its root, and an unlimited number of layers of partial replicas below it. Data is replicated on demand in response to application queries. PathStore supports concurrent object reads and writes on all replicas of the hierarchy; updates are propagated through the hierarchy in the background, providing eventual consistency.

We added session consistency support to PathStore using replica *reconciliation* algorithms that executes when a client switches from a source replica to a destination replica. The basic reconciliation algorithm is simple, yet effective: we track all the items either read or written on the source replica by the session and ensure that the destination has as up to date values by obtaining them from the source. The extension provides read-your-writes and monotonic reads guarantees for clients that switch between otherwise eventual consistent replicas.

We evaluated the performance of our implementation on an emulated multi-layer hierarchical edge deployment. Our results show that session consistency can be enforced as a client switches between arbitrary replicas at a fraction of the bandwidth cost required for maintaining full synchronization even within a small number of replicas. Moreover, our client-reconciliation algorithms dramatically reduces the latency and bandwidth required to enforce session consistency for server applications where a given client only accesses a small fraction of the state stored in a replica.

2 Design Considerations

In this section, we elaborate on our design choices for adding support for session consistency in the context of 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, writes are committed only after they have been stored on all replicas that may handle requests for a session. This approach enables running code on behalf of the same session on multiple replicas concurrently, and supports fast switching between replicas; however, this approach slows down normal operation significantly, may result in high bandwidth consumption and will not work unless all replicas to which the writes need to be propagated are alive and reachable. A reactive implementation ensures session consistency only after a client switches between replicas. Before running code on behalf of a session on a new replica, all relevant data to the session has to be synched with the state available on the last replica. This approach has the disadvantage that a for a given session it can only make use of a single replica at a time. We argue that the reactive approach is more appropriate for edge computing because the replication factor can be high given the large number of edge nodes that a client may switch to. 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 synch using reactive replication is modest.

State between replicas can be synchronized using either *full replication* or *session's data replication*. In the former, the destination replica will have the 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 later, only data relevant to the session, including any records that were read or written, are synced. 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 deployed on the network 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 migration latency, and 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 CQL or SQL to capture access patterns. The benefit of tagging individual records is its simplicity, which comes at the expense of potential significant additional storage overhead. Instead, we opted to track data accesses by recording queries executed against the replica. While this approach is more complex to implement, it has modest storage requirements. In our experience, simple queries can stand in for a large number of rows.

3 Prototype

We developed a distributed storage systems for edge computing that supports session consistency on otherwise eventual consistent replicas. We first describe PathStore [12], the eventual consistent storage system that we use as the basis for our implementation. We then describe extension that add support for session consistency.

3.1 PathStore

PathStore provides a hierarchical eventually consistent database implemented as a tree of independent object stores. A PathStore hierarchy consists of a persistent replica at it root, and an unlimited number of layers of partial replicas below it. PathStore uses Cassandra [11] as its internal storage engine. Each PathStore replica runs a separate independent Cassandra ring, with PathStore in charge of data movement between otherwise independent rings. PathStore replicates data at row granularity on demand in response to application queries. To provide low-latency, all read and write operations are performed against the local replica. PathStore supports concurrent object reads and writes on all replicas of the database hierarchy; updates are propagated through the replica hierarchy in the background. PathStore tags modifications with a version timestamp that records the time the row was inserted, and the ID of the PathStore replica where the modification was originally recorded. PathStore assumes that replicas are tightly synchronized using some accurate mechanism, such as GPS atomic clocks. As modifications propagated through the hierarchy, PathStore uses the version timestamp to determine ordering – most recent timestamp wins.

PathStore's API is based on CQL, Cassandras 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.

3.2 Session Consistency

We extend Pathstore by providing developers the ability to select between eventual and session consistency. We next describe how we group database accesses into sessions, how we track data related to a session, how we synchronize state when a client switches between two replicas, and how we handle failures.

3.2.1 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. The developer can decide to make a session representing a user, a device belonging to a user, 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 unique session id (SID), timestamp, current replica id, 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.

It is up to the the developer to determine how the stoken is communicated between their client and the server code running on the edge datacenter. In our experiments, we use Java Servlets to run our server-side code and pass the stoken using an HTTP cookie.

3.2.2 State Tracking

To keep track of data related to a session, we added to each PathStore replica, a *CommandCache* that stores all the CQL *SELECT* statements run on behalf of a session *s*. The entries in the *CommandCache[s]* precisely identify the data accessed by the *s* since there is a 1-to-1 mapping between query results, and tuples in the database. This is not true in general for SQL, but for CQL, it holds because there are no joins and no aggregation operations (i.e., no *GROUPBY*).

For *INSERT*, *UPDATE* and *DELETE* commands, we also keep track of modified keys affected by these commands using individual *SELECT* queries. For example if the session executes the command where $a1$ is the primary key (*key*):

INSERT INTO pathstore.t1 values (a1,b1)

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

*SELECT * FROM pathstore.t1 WHERE key=a1*

In this way the key can be later accessed using this query. To keep the *CommandCache* smaller, we don't keep queries for a given session that are sub-sumed by more general ones. We also keep queries only for data that is actually replicated by the Pathstore site. Through a background garbage collection mechanism, Pathstore deletes data that has not been accessed for a long time. 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 and every site also keeps track of sessions it is serving.

3.2.3 Switching

PathStore leverages the stoken to detect when a client switches between replicas (e.g., moves between edge replicas n_s, n_d). The new replica (n_d) is able to determine the last location of the session's data by examining the *Current Replica* parameter on the stoken received. Using this information and by identifying the session through the *SID*, the destination replica can initiate the replica switching process.

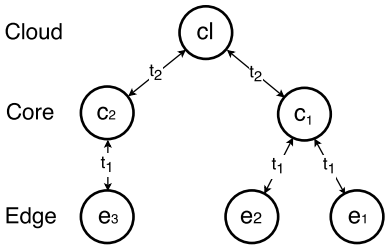


Figure 1: Network topology.

When the switching process for a session is initiated, the *status* field on the stoken changes to *Migrating to* n_d where n_d is the ID of the destination replica. A separate thread then fetches the session's data from n_d. In the meantime if the client moves to 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. To assure session consistency, when a switching process is triggered on n_s, n_s's Pathstore replica will not process further commands for that session. Furthermore, requests for the session are not processed on n_d until the switch is complete.

When a session moves to n_d, the PathStore module of n_d sends a request to the PathStore module of n_s asking for all keys modified or accessed by s. Having recorded all the commands executed by s, n_s's PathStore then re-executes the queries in *CommandCache[s]* to find all modified or accessed rows. It will then put the resulting rows on the PathStore replica of n_d and also move *CommandCache[s]* to n_d's *CommandCache*. Using queries to find accessed rows has the benefit of aggregation. A single query can track many keys read or written by s. Replication is done at full row level irrespective of columns projected in the select query.

3.2.4 Failures

In case of network partitions where the source replicas is unreachable, the application is informed about the issue through an exception. The application can the decide to wait and retry, or invalidate the session and restart.

4 Results

We first compare the performance of PathStore to an alternative deployment of unmodified Cassandra on the edge network shown in Figure 1. We then evaluate the benefits of tracking and only propagating session state when switching between replicas.

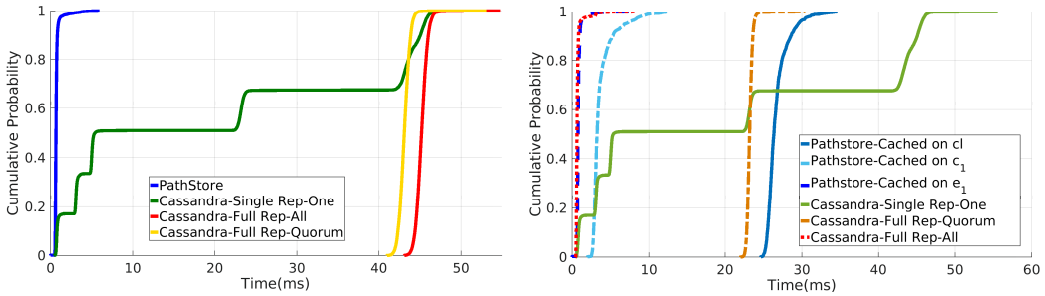
Our experimental setup consists of six servers each acting as a different datacenter. Each server runs an instance of unmodified Cassandra or PathStore, as well as an instance of Apache Tomcat and a simple Java Sevlet that issues CQL queries on either Cassandra or PathStore. The network between the servers is emulated by using Linux's Traffic Control [4], a tool that enabled us to configure the Linux kernel packet scheduler. We assume that the underlay IP network has the same topology (e.g. the round trip time between e_1, e_2 is $2 \times t_1$). We optimistically assume $t_1 = 2ms, t_2 = 20ms$, which tilts the comparison against PathStore and in favor of Cassandra, which is more adversely affected by higher latency.

Figure 2 shows the CDF of the latency for writing or reading a single $1KB$ row on e_1. The experiment is repeated for $10,000$ different rows. The figure shows results for three different PathStore scenarios that assume the rows being read are already replicated on e_1, c_1, and c_l, respectively. There is only one configuration for write in PathStore as all writes are preformed on the local replica (e_1). The figure also shows results for three potential ways in which a developer could attempt to deploy unmodified Cassandra in order to provide a consistent view of the database for a mobile client. In all Cassandra experiments, all six instances of Cassandra (1 per emulated datacenter) are configured as a single ring.

We consider three alternatives [1]: *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. On the other hand, reads can be served from any 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. Finally, *Single Replication-One*, which uses a replication factor of one, and relies on Cassandras standard hashing algorithm to uniformly distribute rows among nodes in the Cassandra ring. Reads and writes in this configuration involve a single server. Table 1 shows the average data transferred aggregated across all links to store or read a $1KB$ row for the various configurations.

All three Cassandra alternatives perform poorly, which is hardly surprising given that Cassandra is not designed to be used in this manner. Conversely, our results are optimistic as real-world edge deployments will likely consist of a much larger number of datacenters. *Full Replication-All* handles reads very well, but pays for it with high latency and bandwidth cost for writes. *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 datacenters. In comparison, PathStore provides low latency for writes and reads, particularly in cases where the row are already

[1] All three Cassandra configuration provide at least serial consistency, which is much stronger that session consistency; however, the stronger properties provide little benefit when the application does not require consistency across different clients.



(a) Writes (b) Reads

Figure 2: CDF of latency required to read and write a $1KB$ row.

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

Table 1: Data transfer.

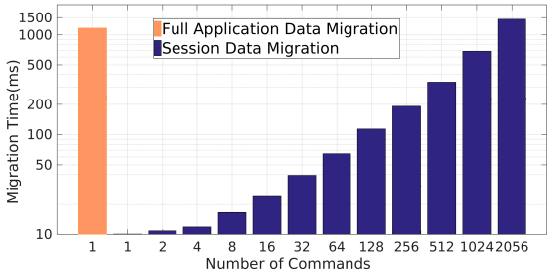


Figure 3: Session migration latency.

available on e_1 or c_1, and requires much less bandwidth.

Figure 3 shows the latency for migrating a session between e_1, e_3. We simulate a server application that in the process of servicing a large number of clients reads 10000 $1KB$ rows from a single table. We show results for two PathStore configurations. The first configuration labeled *Full* does not keep track of data accessed by individual sessions, and as a result has to copy all 10000 when the client moves between replicas. The second configuration labeled *Session Data Migration* uses the *CommandCache* to keep track of rows read by the client that moves between the replicas. We vary the number of commands executed by the client between $1, 2056$ and we assume each command only affects a single row. As shown in Figure 3, for cases where the mobile client accesses only a fraction of the total data used by the service it is 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, executed.

5 Conclusions

A key tenet of edge computing is the ability for clients to be redirected seamlessly across the different edge data centers hosting the replicas of a service, while ensuring a consistent view of the underlying data accessed in the same client session. In this paper, we present a novel storage system that provides session consistency even when the client switches between replicas in different edge locations. Our client reconciliation algorithm enforces session consistency at minimal costs, by tracking the accessed or affected keys and reconciling them on the destination replica. Our results show that our approach provides session consistency at a fraction of the latency and bandwidth costs of a strongly consistent system, and with reasonable migration costs.

In this paper, we have merely taken the initial steps towards the goal of a fully realized system for session consistency at the edge. There are three key directions (among several others) that we plan to pursue to fulfill this goal. First, 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. For example, we can exploit the data propagation of our hierarchical store to transfer data only if the destination is out of date. Second, given the intricate nature of the migrate algorithms, especially in the presence of failures and network partitions, we wish to theoretically analyze and prove the correctness of our algorithms. Finally, we plan to integrate our solution with a prevalent edge use-case (such as an edge file system) and thoroughly evaluate performance.

References

[1] All things distributed: Eventually consistent. https://www.allthingsdistributed.com/2007/12/eventually_consistent.html, 2007.

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

[3] Tunable data consistency levels in azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, 2018.

[4] ALMESBERGER, W. Linux traffic control-implementation overview. Tech. rep., 1998.

[5] BAHL, V. Cloud 2020: The emergence of micro data-centers for mobile computing. *Online: https://www. microsoft. com/en-us/research/wp-content/uploads/2016/11/Micro-Data-Centers-mDCs-for-Mobile-Computing-1. pdf. Accessed 12* (2017).

[6] BONOMI, F., MILITO, R., ZHU, J., AND ADDEPALLI, S. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing* (2012), ACM, pp. 13–16.

[7] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ET AL. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[8] ETSI. ETSI Mobile Edge Computing Working Group. http://www.etsi.org/technologies-clusters/technologies/mobile-edge-computing.

[9] HU, Y. C., PATEL, M., SABELLA, D., SPRECHER, N., AND YOUNG, V. Mobile edge computinga key technology towards 5g. *ETSI white paper 11*, 11 (2015), 1–16.

[10] KAWELL JR, L., BECKHARDT, S., HALVORSEN, T., OZZIE, R., AND GREIF, I. Replicated document management in a group communication system. In *Proceedings of the 1988 ACM conference on Computer-supported cooperative work* (1988), ACM, p. 395.

[11] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[12] MORTAZAVI, S. H., SALEHE, M., GOMES, C. S., PHILLIPS, C., AND DE LARA, E. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (New York, NY, USA, 2017), SEC '17, ACM, pp. 20:1–20:13.

[13] SATYANARAYANAN, M. The emergence of edge computing. *Computer 50*, 1 (2017), 30–39.

[14] SATYANARAYANAN, M., BAHL, P., CACERES, R., AND DAVIES, N. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing 8*, 4 (2009), 14–23.

[15] SHI, W., CAO, J., ZHANG, Q., LI, Y., AND XU, L. Edge computing: Vision and challenges. *IEEE Internet of Things Journal 3*, 5 (2016), 637–646.

[16] SHI, W., AND DUSTDAR, S. The promise of edge computing. *Computer 49*, 5 (2016), 78–81.

[17] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M., THEIMER, M., AND WELCH, B. W. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems* (Washington, DC, USA, 1994), PDIS '94, IEEE Computer Society, pp. 140–149.

[18] VIOTTI, P., AND VUKOLIĆ, M. Consistency in non-transactional distributed storage systems. *ACM Computing Surveys (CSUR) 49*, 1 (2016), 19.

[19] VOGELS, W. Eventually consistent. *Communications of the ACM 52*, 1 (2009), 40–44.