HAL open science

Totally Ordered Replication for Massive Scale Key-Value Stores

José Ribeiro, Nuno Machado, Francisco Maia, Miguel Matos

To cite this version:

José Ribeiro, Nuno Machado, Francisco Maia, Miguel Matos. Totally Ordered Replication for Massive Scale Key-Value Stores. 18th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2018, Madrid, Spain. pp.58-74, 10.1007/978-3-319-93767-0_5 . hal-01824632

HAL Id: hal-01824632

https://inria.hal.science/hal-01824632v1

Submitted on 27 Jun 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Totally Ordered Replication for Massive Scale Key-Value Stores

José Ribeiro[1], Nuno Machado[1], Francisco Maia[1], and Miguel Matos[2]

[1] HASLab – INESC TEC & Universidade do Minho
[2] INESC-ID & Instituto Superior Técnico, Universidade de Lisboa

Abstract. Scalability is one of the most relevant features of today's data management systems. In order to achieve high scalability and availability, recent distributed key-value stores refrain from costly replica coordination when processing requests. However, these systems typically do not perform well under churn. In this paper, we propose DataFlagons, a large-scale key-value store that integrates epidemic dissemination with a probabilistic total order broadcast algorithm. By ensuring that all replicas process requests in the same order, DataFlagons provides probabilistic strong data consistency while achieving high scalability and robustness under churn.

1 Introduction

Distributed key-value stores are widely used nowadays for managing the ever-growing volume of data produced and accessed by Internet services. Key-value stores are appealing due to their high performance, scalability, and availability, but typically do not cope well with *churn* (i.e., the arrival and departure of system nodes of the system) [2, 4, 7]. The ability to handle dynamism is extremely relevant though, as has been already observed in large Internet of Things and blockchain deployments [5].

The main reason why churn heavily affects current key-value systems is that they are built upon structured distributed hash tables (DHTs). DHTs organize nodes in the system into a structured logical overlay that, despite allowing for fast data access on deployments with hundreds of nodes, is not resilient enough to handle high dynamism on larger scale scenarios [11]. In contrast, *unstructured* peer-to-peer (P2P) approaches do not impose any pre-defined structure and hence are robust in the presence of heavy churn.

In the past, we proposed DataFlasks [9], an unstructured key-value store based on epidemic protocols. In DataFlasks, nodes self-divide into groups without the need of global synchronization. Each group is then responsible for handling a certain range of keys, ensuring that queries to those keys are served quickly and that the corresponding data is replicated among the nodes of the group. When peers join or leave the system, the other nodes adjust the group membership in an autonomous fashion and based on local information alone.

Unfortunately, similarly to other NoSQL key-value stores, the performance and scalability benefits of DataFlasks come at the cost of weaker data consistency

guarantees. In detail, it assumes that conflicts are handled at the client side and does not offer any concurrency control mechanism. This means that two concurrent writes to the same key, with the same version but different values, may lead to an inconsistent scenario in which nodes of the same group store distinct data. In other words, the replicas diverge.

This paper aims at moving a step forward towards the design of a large-scale data management system capable of simultaneously offering high scalability, robustness, and data consistency. We propose a novel key-value store architecture that stems from the integration of DataFlasks with EpTO [10], a probabilistic total order broadcast algorithm. We first show that this architecture can be easily adapted to provide different trade-offs between performance and consistency. Then, we explore a particular instance of such trade-offs with DataFlagons[3]. In a nutshell, DataFlagons is a key-value store that shares the scalability, availability, and flexibility benefits of DataFlasks while providing strongly consistent and reliable replication (as given by EpTO). Furthermore, DataFlagons obviates the need to control versioning at the application level, thus offering a simpler programming interface. Our experimental evaluation based on simulations shows that DataFlagons is scalable, resilient to churn, and that it successfully overcomes the data replication issues observed in DataFlasks, albeit at the cost of 4.7x throughput slowdown.

The rest of the paper is organized as follows. Section 2 overviews the related work, as well as DataFlasks and EpTO. Section 3 describes the design and implementation of DataFlagons. Section 4 presents the experimental evaluation. Finally, Section 5 concludes the paper by summarizing its main findings.

2 Background and Motivation

2.1 State of the Art

Over the last decade, there has been a large body of work on distributed data stores with a key-value model. The most relevant examples are arguably Cassandra [7], Dynamo [4]/Voldemort[4], Bigtable [1]/HBase[5], and PNUTS [2]. These systems explore different compromises between consistency, partition tolerance, and availability, although most choose to sacrifice strong consistency guarantees for higher performance and scalability. For data storage and retrieval, Cassandra and Dynamo rely on a DHT-based structure, PNUTS uses a modified B+ tree, and Bigtable relies on a set of master nodes. This structured design, despite scaling up to hundreds of nodes, typically lacks *robustness* in environments subject to churn. Here, robustness can be defined as the ability to cope with errors during execution and becomes particularly relevant as the system scales and the workload increases [8].

[3] Our system was named DataFlagons to convey an improvement over DataFlasks, as flagons are arguably more robust and consistent containers than flasks.

[4] http://www.project-voldemort.com/voldemort/

[5] https://hbase.apache.org

On the other hand, DataFlasks [9] performs well even in highly dynamic massive scale settings by operating on top of unstructured peer-to-peer protocols. However, it has the drawback of lacking consistency guarantees during data replication, which can cause replicas to diverge.

Figure 1 sketches a generic comparison of the aforementioned systems in terms of their robustness and consistency properties [8].[6] From the figure, one can observe that there is currently a gap in terms of robust solutions, capable of coping with high churn, that ensure data consistency. In this paper, we aim at filling this gap with a novel key-value store architecture that combines DataFlasks with EpTO [10], a probabilistic total order broadcast algorithm. The resulting architecture benefits from the resilience and scalability of DataFlasks, but is able to provide stronger consistency guarantees via total order delivery. Next, we describe DataFlasks and EpTO.

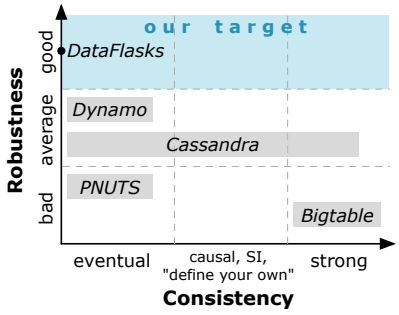

Fig. 1. Qualitative comparison of prior key-value stores in terms of (single-tuple) consistency and robustness.

2.2 DataFlasks

DataFlasks [9] is a distributed key-value store that stores tuples of format (K, V, v), where K indicates the key mapping to a value V with version v. Clients can add tuples to the system by invoking $put(K, V, v)$, and retrieve the value V using the operation $get(K, v)$. Any DataFlasks node can handle client requests. Figure 2 depicts the architecture of DataFlasks in more detail. As shown in the figure, DataFlasks is composed of four main modules: *request handler*, *group construction*, *anti-entropy*, and *dissemination*, which we describe next.

Request handler. Upon receiving a *put* request, a node first checks whether the group it belongs to is responsible for handling that particular key. This is done by checking whether the tuple's key falls within the group's key range. If so, the node stores the tuple and forwards the request to its group peers for replication purposes. Otherwise, the request is forwarded to other groups via epidemic dissemination.

In turn, when a node receives a *get* of a given key and version, it immediately replies to the client in case it holds the corresponding value. Otherwise, the node simply disseminates the request to the other groups, which will be in turn responsible for sending the value to the client.

Group construction. This component is responsible for organizing nodes into groups of *at least* the size of the desired replication factor. This is done by continuously estimating the number of groups that should be maintained according

[6] Note that this is a high-level quantitive comparison based on prior studies [8], as a thorough experimental analysis for robustness is currently lacking in the literature.

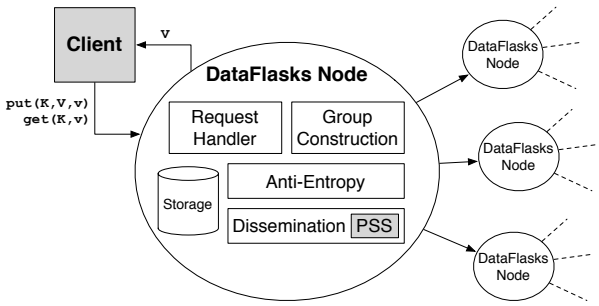

Fig. 2. DataFlasks architecture. The system returns a value `V` in response to *get* operations. For *put* operations, DataFlasks simply returns an acknowledgement.

to the system size. Under churn, nodes leaving the system cause the estimated number of groups to decrease, while nodes joining the system lead to the creation of new groups, thus maintaining the stability of the group size.

Anti-entropy. This module periodically contacts another node from the same group to ensure that all tuples were correctly replicated and no data is missing.

Dissemination. This component is responsible for node communication via gossip. Concretely, each node in DataFlasks resorts to a *peer sampling service (PSS)*, namely Cyclon [13], to obtain a *partial view* of the system, containing references to neighbor peers. The union of all nodes' partial views creates a logical overlay that evolves with time and allows efficient dissemination of information.

Local storage is a component that abstracts how data is persisted. A more thorough description of DataFlasks' design can be found in [9].

The case for total order. DataFlasks has no concurrency control mechanism. Instead, it relies on the clients to version each operation. If clients do not take this into account, scenarios of replica state divergence may occur. To provide further evidence of this issue, we implemented the algorithm of DataFlasks in Python to perform an experiment.[7] It consisted in having two clients concurrently inserting different values – $v1$ and $v2$ – by executing $put(k, v1, 0)$ and $put(k, v2, 0)$, respectively. Note that both tuples have the same key and version, thus represent the same data item in DataFlasks. For this experiment, DataFlasks ran on 300 nodes distributed among 32 groups.

Figure 3a reports the number of replicas of each value existing in the group responsible for handling key k, for a simulation of 8000 time units and observations performed every 125 time units. The figure shows that the two values were successfully replicated by disjoint subsets of nodes of the group and that no state changes were observed until the end of the simulation. Since DataFlasks employs a *first-write-wins* approach, once a node receives the first *put* operation referring to key k and version 0, it will ignore subsequent writes for the same key and version. Figure 3b confirms this fact by plotting the instant in which each *put* arrived to two distinct nodes, denoted X and Y. Given that replicas X and Y

[7] Our implementation of DataFlasks, as well as that of the simulator used in the experiments, are detailed in Section 4.

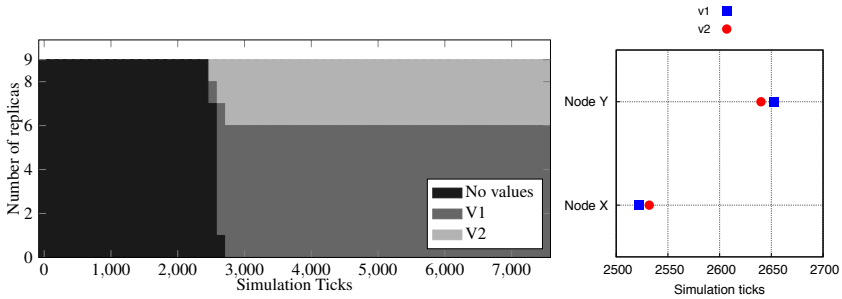

Fig. 3. (a) Node states in DataFlasks when concurrently storing values $v1$ and $v2$ for the same K, v. (b) Detail with request arrival order at two different nodes that leads to state divergence.

receive $v1$ and $v2$ in opposite orders, both values end up being persisted and the state of the system diverges. This inconsistency is problematic because it allows read operations for the same tuple to return different values. On the other hand, ensuring that the requests have different versions (i.e. the requests are ordered) would suffice to overcome this drawback of DataFlasks. DataFlagons follows this approach with the help of a total order broadcast algorithm, which we describe next.

2.3 EpTO

EpTO [10] is a scalable total order algorithm with probabilistic guarantees of total order delivery. This is done by having nodes agree (with high probability) on the order in which events received should be delivered to the application. Unlike traditional agreement algorithms though, EpTO achieves this property in a fully decentralized and coordination-free manner, which allows for high scalability and robustness in face of churn and message losses.

EpTO relies on an adaption of the *balls-and-bins* [6] algorithm to epidemic dissemination, where nodes are represented as *bins* and messages as *balls*. The main goal of the algorithm is then to send as many balls as necessary such that all bins receive at least a ball with high probability. To meet this goal, a node wishing to disseminate a ball (i.e. a set of messages) across the system simply sends it to K of its neighbors, chosen uniformly at random. In the following rounds, the nodes that received the ball will propagate it to another K random neighbors, thus quickly spreading the rumor through the network. After a few rounds, all the nodes have received all the messages with high probability. Hence, EpTO can then ensure total order delivery by simply sorting the messages according to a deterministic factor.

EpTO exposes two primitives to the application, namely *EpTO-broadcast* and *EpTO-deliver*, which are guaranteed to satisfy the following properties:

– **Integrity.** For every message m, each node *EpTO-deliver*s m at most once, iff m was previously *EpTO-broadcast*ed.

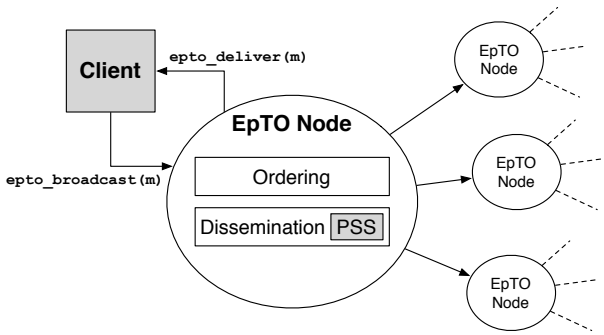

Fig. 4. EpTO architecture. `m` represents a message being broadcast/delivered by EpTO.

– **Validity.** If a correct node (i.e. a node that did not fail or left the system) *EpTO-broadcast*s a message m, then it will eventually *EpTO-deliver* m.

– **Total Order.** If two nodes P and Q *EpTO-deliver* two messages m and m', then P *EpTO-deliver*s m before m' iff Q *EpTO-deliver*s m before m'.

– **Probabilistic Agreement.** If a node *EpTO-deliver*s a message m, then, with high probability, all correct nodes will eventually *EpTO-deliver* m.

Figure 4 illustrates EpTO's architecture. It comprises a *dissemination* component, which disseminates events, and an *ordering* component, which delivers events to the application in total order. The total order is satisfied by the ordering component, whereas the probabilistic agreement is satisfied by the dissemination component. The properties of integrity and validity are guaranteed by the two components in conjunction.

The dissemination component operates in periodic rounds of δ time units and aims at broadcasting balls across the network in a gossip-based fashion. As in DataFlasks, nodes in EpTO rely on an underlying PSS to obtain a fresh and randomized partial view of the system. This view is assumed to contain at least K correct nodes to which balls are propagated, denoted as the node's *fanout*. Each ball should be relayed for a *time to live (TTL)* number of rounds, with TTL being a configurable parameter of the system. Each node stores the messages to be forwarded in a set denoted *nextBall*.

When a node wishes to broadcast a message, it sets the message's *ttl* to 0, assigns a timestamp (i.e. a logical clock), and appends it to the *nextBall* set. In turn, when a node receives a ball of messages, it compares each message's *ttl* against the TTL configuration parameter. If $ttl < TTL$, the incoming message is added to *nextBall* for future retransmission. In case there is a message with the same id already present, EpTO keeps the one with higher *ttl*.

Independently to message broadcast and reception, at the beginning of every round (i.e. every δ time units), each node *i)* increments by one the *ttl* of the messages stored in *nextBall*, and *ii)* sends this set as a ball to K random neighbors. EpTO's execution flow then proceeds to the ordering component.

To ensure total order delivery, the ordering component maintains two lists of messages: one storing the messages received up to that moment and the other

containing the messages that were already delivered to the application. The ordering component's main task is then to move messages from the list of *received* to the list of *delivered*, while avoiding duplicates and preserving the global order. Messages in the received list that were already delivered to the application are discarded, whereas the remaining ones are only added to the delivered list if their *ttl* already reached the TTL value. For this latter case, EpTO then delivers the messages to the application in ascending order of their timestamp, using the id of the broadcaster node to break ties.

Note that the probabilistic nature of the agreement property offered by EpTO permits a rare yet possible scenario in which some messages are not delivered at all by a subset of nodes. These nodes, despite exhibiting holes in the sequence of messages agreed upon, are guaranteed to preserve the total order property of message delivery to the application. In other words, the total order is ensured but the agreement might be violated (with arbitrarily low probability).

A keen observation of the architectures of DataFlasks and EpTO (in Figure 2 and Figure 4, respectively) is that the two systems exhibit interesting similarities: both organize nodes into an unstructured, logical overlay and both resort to a PSS to disseminate events. These similarities paved the way for the design of DataFlagons and its properties, as described in the next section.

3 DataFlagons

DataFlagons is designed as an integration of DataFlasks with EpTO, inheriting components and properties from both systems. This integration yields a synergy that allows not only obtaining the best of both solutions but also addressing the shortcomings of each one of them individually. On the one hand, the total order property of EpTO solves the replica divergence issue of DataFlasks discussed before. As all nodes in DataFlagons execute all operations by the same order, it becomes impossible for different replicas to store distinct values for the same key and version. Alongside, using total order in DataFlagons strengthens the poor consistency guarantees originally offered by DataFlasks.

On the other hand, DataFlasks' anti-entropy mechanism helps to cope with the potential holes existing in EpTO's delivery. In the following, we describe the architecture of DataFlagons and discuss the trade-offs between consistency and throughput that it enables.

3.1 System Overview

DataFlagons is a scalable, robust, and strongly consistent key-value store. To the best of our knowledge, DataFlagons is the first system capable of providing these properties in massive scale environments subject to churn. Figure 5 illustrates DataFlagons' architecture, indicating the components from both DataFlasks and EpTO. As shown, DataFlagons inherits the request handler, the group construction, the anti-entropy, and the storage components from the former. From the latter, DataFlagons inherits the ordering and dissemination modules.

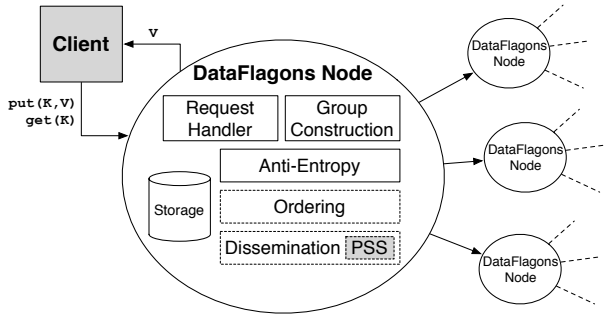

Fig. 5. DataFlagons architecture as an integration of components from DataFlasks (solid line boxes) and EpTO (dashed line boxes).

As a concrete example of the *modus operandi* of DataFlagons, consider the diagram of Figure 6, representing the execution flow of a *put* in our system.[8]

① A client wishing to add new tuples to the data store has simply to issue a *put* request indicating the key and the value. This contrasts to DataFlasks, where it was necessary to specify the version in addition to the key and the value. Likewise, for performing a *get* operation in DataFlagons, the client needs only to indicate the key it wants to read, instead of the key and the version.

② The incoming request is processed by DataFlagons' request handler. In practice, this corresponds to encapsulating the operation in a *ball* to be disseminated through the network via *EpTO-broadcast*.

③ Following EpTO's algorithm, the invocation of the *EpTO-broadcast* primitive results in the propagation of the ball to K random neighbors of the broadcasting node. The neighbors receiving the ball will later forward it to another K peers of their own view.

④ In the following rounds, the node will receive, with high probability, balls containing the message previously broadcast. All messages received are forwarded to the ordering component.

⑤ Upon the reception of a message with a *ttl* equal to the TTL of the system, the ordering component checks if the event's timestamp is the next in line to be delivered according to the total order.[9] If it is, the ordering component executes the *EpTO-deliver* primitive. However, unlike in EpTO, the component notified by this primitive in DataFlagons is the request handler and not the client, as the processing of the actual operation (i.e. the *put*) has not been concluded yet.

⑥ When the request handler receives an operation via total order delivery, it proceeds as in DataFlasks and queries the group construction component in order to assess whether the tuple should be stored locally.

⑦ The group construction, in turn, checks whether the key belongs to the group's key range. If so, the tuple is persisted in the storage; otherwise, it is dis-

[8] We omit a description of the flow for a *get* request, as it is similar to that of a *put*.

[9] For messages with the same logical clock, DataFlagons orders them in ascending order of the ids of their broadcasting nodes.

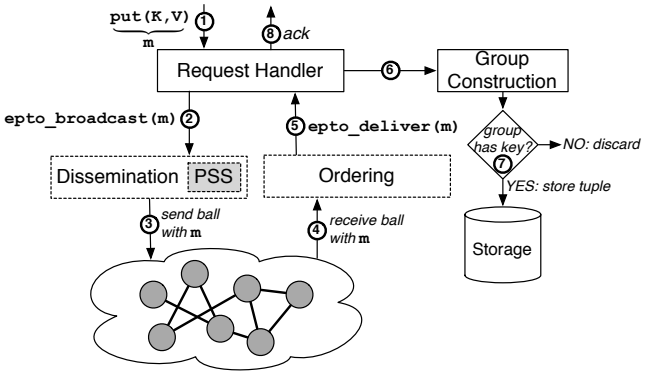

Fig. 6. Execution flow of a *put* operation in DataFlagons. This operation is abstracted as a message `m` disseminated via EpTO.

carded. Since the dissemination mechanism of EpTO already delivers the events to all nodes with high probability, DataFlagons no longer needs to propagate the operation to other groups in case the key is not the responsibility of the node's group, as opposed to DataFlasks.

⑧ Finally, DataFlagons acknowledges the client that the operation was successfully performed.

Note that the anti-entropy module is not included in the diagram of Figure 6. The reason is that anti-entropy operates orthogonally to the common request handling procedure, ensuring that all nodes of the group have a replica of the data despite potential holes in EpTO's dissemination. Anti-entropy is also particularly helpful to secure data durability and improve system availability during periods of churn. When a new node joins DataFlagons, this module ensures that it quickly receives, from another peer of the same group, a replica of the data for which it is responsible.

3.2 Design Trade-offs

Recall the robustness-consistency spectrum depicted in Figure 1. The design of DataFlagons has the advantage of allowing us to easily support various consistency levels while maintaining the system highly robust. This is achieved by tuning the module through which *get* and *put* requests are propagated across the network. Table 1 summarizes these design choices and their respective consistency level.

As shown in the table, using DataFlasks alone to disseminate the two types of requests naturally causes DataFlagons to lose any data consistency guarantees. In contrast, if we rely on DataFlasks to only propagate *get*s and use EpTO to broadcast *put*s, the consistency level increases to *eventual*. A system is *eventually consistent* if it guarantees that all replicas will eventually return the last updated value of an item, assuming that no new updates are made to that item [12]. In other words, there is a time window in which two concurrent reads on different

Consistency Level	Dissemination Module	
	get	*put*
No guarantees	DataFlasks	DataFlasks
Eventual	DataFlasks	EpTO
Strong (probabilistic)	EpTO	EpTO

Table 1. Consistency levels offered depending on the dissemination module used by DataFlagons. The shaded row indicates the design choice explored in this paper.

replicas can return different values for the same key, although both replicas will eventually converge to the same value.

Finally, delivering both *get* and *put* requests in total order via EpTO renders DataFlagons probabilistically strongly consistent. *Strong consistency* states that any read happening after an update will return the updated value [12]. Although we believe that DataFlagons is strongly consistent for the vast majority of scenarios, we claim that it offers probabilistic strong consistency due to the very unlikely yet possible event where a subset of nodes does not deliver all the messages. This event is equivalent to not applying all operations, which would allow a small window of inconsistency. This issue can be solved but it is out of the scope of the present paper.

For space constraints, in this paper, we only explore propagation done exclusively through EpTO which provides stronger consistency. As advocated by prior work [3], strong consistency has the advantage of significantly simplifying the implementation of applications and reduce the occurrence of bugs. Nevertheless, preliminary results about the other design choices show that customizable consistency is a promising approach.

The next section presents an experimental evaluation of DataFlagons with probabilistic strong consistency by comparing it against a vanilla version of DataFlasks.

4 Evaluation

We implemented a prototype of DataFlagons in Python and evaluated it by means of simulations. In particular, our experimental analysis focuses on answering the following questions:

– **Consistency:** Does DataFlagons ensure consistent replication? (§4.1)

– **Performance:** How does DataFlagons compare to DataFlasks in terms of response latency, throughput, and the number of messages when subject to different churn levels? (§4.2)

The experiments were conducted on a machine with a 3.4 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk. The discrete simulator used was that of EpTO's evaluation [10], which allows easily tuning the number of nodes in the system, network latency, churn, message loss, and the duration of the simulation.[10] The passage of time is modeled as simulation ticks by means of a priority queue and a monotonically increasing integer.

[10] https://github.com/miguelammatos/SimpleDA

Regarding the experimental setting, the network size was varied between 300 and 1000 nodes, and the end-to-end latency was parameterized according to real-world PlanetLab traces as in Matos *et al.* [10].

For the configuration of DataFlagons, we used the values suggested in Matos *et al.* [10] and Maia *et al.* [9]: EpTO's fanout and TTL were set as 18 and 25 respectively, while DataFlasks' group construction boundaries were set as $[6, 12]$ (i.e., each group much have at least 6 replicas and at most 12 replicas). In all experiments, we initialize the PSS of each node with a view containing 20 references to other nodes, chosen uniformly at random, and with a shuffling period of 125 simulation ticks. This allows DataFlagons to be bootstrapped on top of a random overlay. The anti-entropy mechanism of DataFlasks and the relaying procedure of EpTO also execute every 125 ticks.

Finally, at the beginning of each simulation, we first let DataFlagons run until convergence, meaning that all nodes have the correct estimation of both the total number of groups in the system and to which group they belong. After that, the experiment's respective workload is executed.

4.1 Consistency Results

To thoroughly assess DataFlagons' data consistency guarantees, we conducted an experiment similar to that of Section 2.2, in which two clients concurrently write different values for the same key. However, in this case, the experience is more elaborate in that the two clients perform 20 requests each (80% *gets* and 20% *puts*) for churn levels of 10%, 20%, and 30%. The goal of the experiment is twofold: *i)* check whether DataFlagons delivers the requests in order, and *ii)* confirm that strong consistency is guaranteed under churn.

To ensure that the operations are handled by the system despite node failures, each request is sent to three nodes randomly chosen. For *get* requests, the value read is considered to be that of the first response.

Churn is implemented by replacing an existing node with a fresh one, ensuring that the distribution of nodes leaving/entering the system is balanced across all groups. As such, a test case with a churn level of 10% indicates that each group is subject to 10% of churn. The churn period was configured in order to allow the convergence of the system in-between membership variations. When a new node joins the system, it also receives a random partial view with references to the nodes alive at that moment.

Figure 7 reports the results of the experiments for a system with 300 nodes and simulations of 32K ticks. In particular, the figure depicts the total order of requests performed by DataFlagons (note that each *put* has its own unique version) and the version of the key returned by each *get*, which corresponds to a unique value. In this context, a violation of the strong consistency guarantees would be observed as a mismatch between the version read by a *get* request and the version of the latest *put* delivered by DataFlagons. The results in the figure show that, for all churn levels, no consistency violations occurred and that DataFlagons successfully replicated the values across all nodes of the group

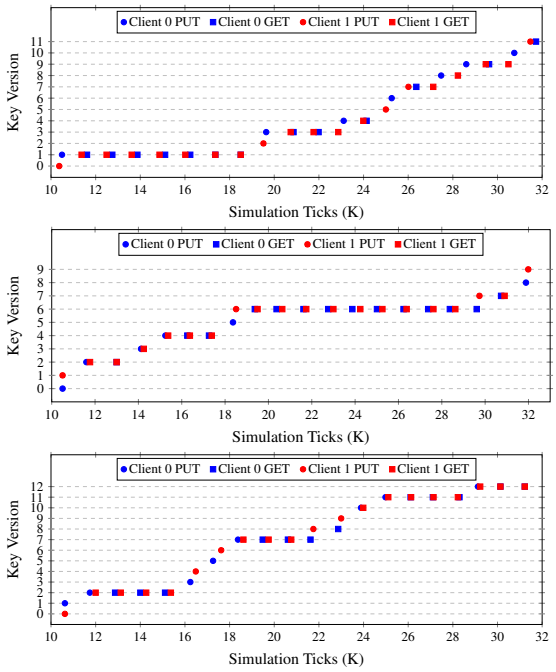

Fig. 7. Request delivery order and consequent versioning given by DataFlagons for churn of 10%, 20%, and 30% (top to bottom) applied to a system of 300 nodes. No violations of strong consistency were observed, as *get* operations always returned the value written by the latest *put*.

without state divergence. This outcome corroborates the effectiveness of EpTO in delivering events in total order.

4.2 Performance Results

We now assess the performance of DataFlagons, by comparing its latency, throughput, and number of messages exchanged against those of DataFlasks. The workload used for these experiments consists of a number of clients (namely, 30 and 100 clients respectively for the scenarios with 300 and 1000 nodes) systematically performing requests until the end of the simulation, with a *get*/*put* ratio of 80%/20%. Naturally, the operations on DataFlasks are associated with a version, as opposed to the operations on DataFlagons.

Clients issue each request to a randomly chosen node and receive acknowledgments from all the nodes belonging to the group responsible for handling that key. Since the way one treats the acknowledgments poses a trade-off between throughput and data durability, we opted for conducting tests with two different versions of DataFlasks exploring different compromises:

– **DFlasks-1/3.** A request is only considered complete after the client receives acknowledgments from one-third of the average number of replicas of the

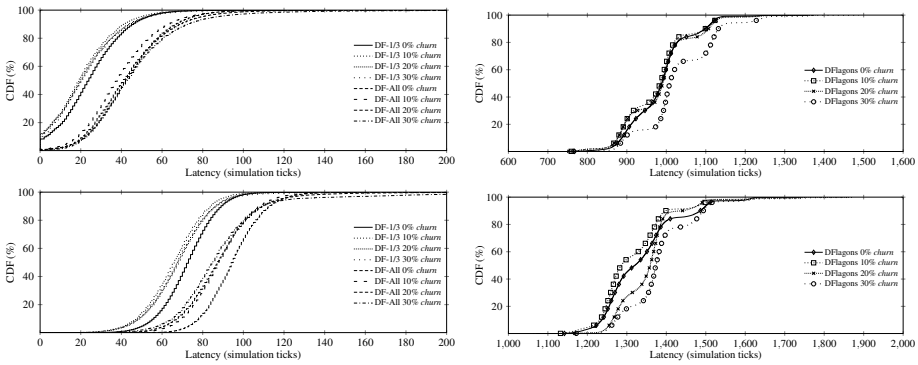

Fig. 8. Response latency of the three systems for scenarios with 300 nodes (top row) and 1000 nodes (bottom row), and churn levels of 0%, 10%, 20%, and 30%. For readability purposes, the results for each simulation were divided into two different plots (depicted in the same row), corresponding to the relevant portions of the experiment.

group. For the group construction interval $[6, 12]$ considered in the experiments, it means that the client has to receive 3 acknowledgments before being able to issue another request ($(6 + 12)/2/3 = 3$).

– **DFlasks-All.** A request is only considered complete after the client receives acknowledgments from a number of replicas equal to the lower bound of the group construction interval. For the interval $[6, 12]$, it means that the client has to receive 6 acknowledgments before being able to issue another request.

Note that, for DataFlagons, the client only needs to receive a single acknowledgment to achieve durability, because the request is guaranteed to have been previously broadcast through the network via *EpTO-broadcast*.

Our experiments also considered different churn levels, namely 0%, 10%, 20%, and 30%. As in the previous section, churn is implemented by replacing an existing node by a fresh one, ensuring that the distribution of nodes leaving/entering the system is balanced across all groups. The results of the experiments, averaged from five runs, are as follows.

Response Latency. Figure 8 depicts the *cumulative distribution function (CDF)* of the response latencies of DataFlagons, DFlasks-1/3, and DFlasks-All, measured for the different testing scenarios. The main observation stemming from the results is that DataFlagons incurs significantly higher latency than both DataFlasks configurations, being one order of magnitude slower. The slowdown in DataFlagons is due to the time required by EpTO to achieve probabilistic agreement and, consequently, enable total order delivery. In contrast, replicas in DataFlasks can reply to the client immediately after receiving the request.

Regarding the robustness and scalability of DataFlagons, the results indicate that there is a slight increase in the response latency as the churn and system size grow (of up to 10% and 27%, respectively). Since the slowdown is not substantial, we argue that the experiments reveal that DataFlagons is robust and scalable.

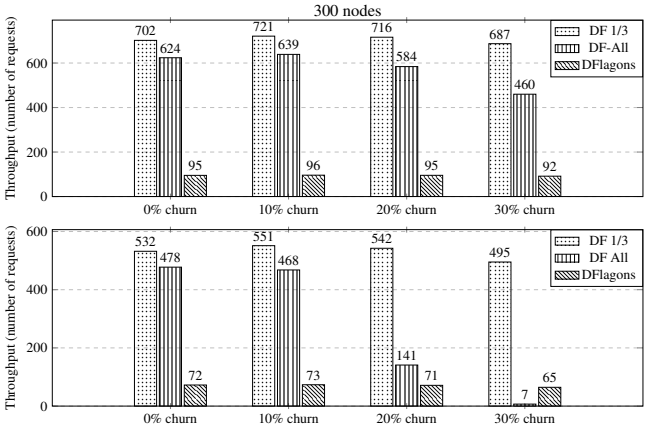

Fig. 9. Throughput of the three systems for scenarios with 300 nodes (top) and 1000 nodes (bottom), and churn levels of 0%, 10%, 20%, and 30%.

Throughput. Figure 9 shows the throughput of DataFlagons, DFlasks-1/3, and DFlasks-All observed in the experiments. These results follow the same trend as those for response latency: DFlasks-1/3 exhibits the highest throughput, closely followed by DFlasks-All, while DataFlagons processes an order of magnitude fewer requests. Figure 9 also demonstrates that increasing churn barely affects DataFlagons, whereas DFlasks-All is heavily hampered. Since this configuration of DataFlasks requires all replicas to answer to the client, having more nodes leaving the network will necessarily reduce the system's ability to process requests, potentially leading to clients becoming blocked when the system fails to maintain the minimum number of replicas (for 1000 nodes with 30% churn).

Message Cost. We measured the average message cost per request for the different systems, which includes the messages exchanged by the anti-entropy mechanism, the peer sampling service, and EpTO's relaying (for DataFlagons). For the scenario with 300 nodes, DFlasks-1/3, DFlasks-All, and DataFlagons required, on average, 200K, 200K, and 4M messages to process a request, respectively. In turn, for 1000 nodes, the message overhead increased respectively to 2M, 2M, and 60M. Finally, we did not observe significant variations of the number of messages exchanged when increasing churn.

Improving DataFlagons' performance by parameter tuning. The aforementioned experiments reveal that the stronger consistency guarantees offered by DataFlagons come at the cost of around an order of magnitude performance slowdown comparing to DataFlasks. However, since the performance of DataFlagons is directly related to the conservativeness of EpTO's parameters, we performed a sensitivity analysis in order to assess the throughput improvements achievable by our system when varying the fanout and TTL. The analysis is omitted due to space constraints, but the outcomes indicate that, for the same message overhead as DataFlasks and without compromising strong consistency,

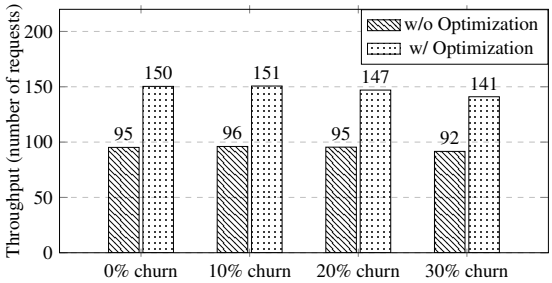

Fig. 10. Throughput of DataFlagons after fine-tuning the fanout and TTL versus that of a non-optimized version, for a scenario with 300 nodes.

DataFlagons can improve its performance using a fanout of 11 and a TTL of 8. Figure 10 plots the throughput of the new configuration for 300 nodes.

The results in the figure show that fine-tuning the parameters of EpTO leads to an average throughput increase of 58% with respect to the original configuration of DataFlagons. Again, it should be noted that this parameter tuning did not affect the consistency guarantees offered by DataFlagons.

We also note that the flexible design of DataFlagons permits further performance improvements through the exploration of weaker data consistency guarantees (see Section 3.2). For instance, by applying EpTO's dissemination solely to *put* requests within the nodes of the same group while propagating *get*s via DataFlasks' traditional gossip mechanism, one could achieve substantial throughput benefits at the cost of lowering the consistency level from strong to eventual. We believe that the exploration of these trade-offs on DataFlagons poses an interesting research direction, which we intend to pursue in the future.

5 Conclusions

This paper addresses the issue of efficiently storing and managing data in massive-scale environments subject to churn. In particular, we propose DataFlagons, a scalable and probabilistically strongly-consistent distributed data store that stems from the combination of DataFlasks (a large-scale key-value store) with EpTO (a probabilistic total order broadcast algorithm). The experimental evaluation of DataFlagons based on simulations shows that it is able to address the consistency issues observed in DataFlasks with the same scalability and availability, albeit reducing the performance on 4.7x on average.

This is, to the best of our knowledge, the first system ever to offer simultaneously strong consistency guarantees coupled with the ability to handle high levels of churn for systems with thousands of nodes.

Acknowledgments. The authors would like to thank the anonymous reviewers for their valuable feedback. This work was partially supported by Project "TEC4Growth - Pervasive Intelligence, Enhancers and Proofs of Concept with

Industrial Impact" (NORTE-01-0145-FEDER-000020), financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project "POCI-01-0145-FEDER-006961", and by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência as part of project "UID/EEA/50014/2013".

References

1. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2) (Jun 2008)
2. Cooper, B.F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.A., Puz, N., Weaver, D., Yerneni, R.: Pnuts: Yahoo!'s hosted data serving platform. Proc. VLDB Endow. 1(2), 1277–1288 (Aug 2008)
3. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J.J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., Woodford, D.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. 31(3) (Aug 2013)
4. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. ACM SIGOPS operating systems review 41(6) (2007)
5. Eyal, I., Gencer, A.E., Sirer, E.G., Van Renesse, R.: Bitcoin-ng: A scalable blockchain protocol. In: NSDI '16. USENIX Association (2016)
6. Koldehofe, B.: Simple gossiping with balls and bins. Stud. Inform. Univ. 3(1) (2004)
7. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review 44(2) (2010)
8. Lourenço, J.R., Cabral, B., Carreiro, P., Vieira, M., Bernardino, J.: Choosing the right nosql database for the job: a quality attribute evaluation. Journal of Big Data 2(1), 18 (Aug 2015)
9. Maia, F., Matos, M., Vilaça, R., Pereira, J., Oliveira, R., Riviere, E.: Dataflasks: epidemic store for massive scale systems. In: SRDS'14. IEEE (2014)
10. Matos, M., Mercier, H., Felber, P., Oliveira, R., Pereira, J.: EpTO: An epidemic total order algorithm for large-scale distributed systems. In: Middleware'15. ACM (2015)
11. Rhea, S., Geels, D., Roscoe, T., Kubiatowicz, J.: Handling churn in a DHT. In: Proceedings of the Annual Conference on USENIX Annual Technical Conference. pp. 10–10. ATC '04, USENIX Association, Berkeley, CA, USA (2004)
12. Vogels, W.: Eventually consistent. Commun. ACM 52(1) (Jan 2009)
13. Voulgaris, S., Gavidia, D., van Steen, M.: Cyclon: Inexpensive membership management for unstructured p2p overlays. Journal of Network and Systems Management 13(2) (2005)