DDFLASKS: deduplicated very large scale data store

Francisco Maia, João Paulo, Fábio Coelho, Francisco Neves, José Pereira, Rui Oliveira
High Assurance Software Laboratory
INESC TEC and UMinho
{*fmaia, jtpaulo*}@*di.uminho.pt*, {*fabio.a.coelho, francisco.t.neves*}@*inesctec.pt*, {jop, rco}@di.uminho.pt

Submission Type: Research

Abstract

With the increasing number of connected devices, it becomes essential to find novel data management solutions that can leverage their computational and storage capabilities. However, developing very large scale data management systems still requires tackling a number of interesting distributed systems challenges. Namely, the intrinsic dynamism observed in large scale systems demands software that is able to cope with continuous failures and high levels of node churn. In this context, pro-active approaches to fault tolerance proved suitable and effective. In particular, epidemic-based protocols are known for their resilience and have been successfully used to build DATAFLASKS, an epidemic data store for massive scale systems. Notwithstanding the fact that it is able to cope with very high levels of churn, the underlying pro-active approach to fault tolerance is resource demanding.

In this paper we extend our epidemic data store with deduplication to design DDFLASKS. This novel system is evaluated in a real world scenario using several Wikipedia snapshots, and the results are twofold. First, we show that deduplication is able to decrease storage consumption from 45% up to 63%. Second, we show that deduplication is also able to decrease network bandwidth consumption by up to 20%. These improvements are key for increasing the effectiveness and applicability of our very large scale data store.

The system developed is open-source and, to the best of our knowledge, is the only data store tailored for massive scale systems and with deduplication built-in.

1 Introduction

For many years now we hear promises of the emergence of the Internet of Things (IoT) and of Edge Computing. Still, the idea of a world of interconnected things has remained more an idea than a concrete reality. Recent predictions from the International Data Corporation (IDC) studies, however, point to significant developments in this area and it is expected that by 2020 there will be an extraordinary number of 32 billion things connected to the Internet [17]. Additionally, IDC studies also point that the amount of digital data will grow from 4.4 Zettabytes in 2013 to 44 Zettabytes in 2020.

Naturally, an explosion in the number of connected devices and in the amount of data being produced and exchanged demands for novel approaches to data management. Massive scale systems, composed of thousands to millions of devices, exhibit specific characteristics that are specially challenging and need to be addressed. Namely, the increase in scale is necessarily accompanied by an increase in system dynamism. Such dynamism arises both from failures that, in these environments, become the rule instead of the exception and by the natural constant entrance and departure of devices, which we will call nodes from now on [28].

Alongside, real world applications start to struggle to find affordable systems to manage and store massive amounts of data. As an example, the Wikimedia Foundation is currently requesting help to users that have spare storage and bandwidth capabilities to store and host Wikipedia snapshots [12]. These snapshots contain the entire history of Wikipedia across distinct periods of time and are valuable for a wide variety of users including researchers. In fact, it is, nowadays, really hard, if not impossible, for common users to have access to older Wikipedia snapshots as Wikimedia has limited storage capabilities. Offering a massive scale storage system able to accommodate the entire Wikipedia and its history relying only on commodity hardware is of significant interest. Moreover, serving all these snapshots from an unified storage service, instead of scattering the snapshots across independent storage systems, is key for users to have an efficient way of accessing the full history of Wikipedia.

Recent research work proposed a data store entirely built with epidemic protocols, tailored precisely for large scale environments [22]. The success of DATAFLASKS, with respect to coping with high levels of system dynamism, lies in its autonomous and unstructured approach

to node organization and in its pro-active approach to fault tolerance. In DATAFLASKS, nodes autonomously organize themselves into groups that are responsible for a subset/partition of the data. Then, the number of nodes in a group determines the data replication factor for the data being stored. Within each group, nodes periodically and proactively contact each other in order to maintain desirable data replication levels. This architecture is simple enough to remain manageable and elegant while, at the same time, is conveniently flexible. Note that, increasing the number of groups increases the storage capacity of the store while increasing the number of nodes within a group increases the replication factor for the data being stored.

The effectiveness of a pro-active approach to data replication comes, unfortunately, with an increase in storage and network resource usage. In particular, periodic and pro-active exchange of data between nodes yields constant network bandwidth consumption even when the system is in a moderately stable period *i.e.*, with a low churn ratio. In fact, bandwidth is actually a bottleneck for scalability in this type of systems and, even though DATAFLASKS autonomous data partitioning alleviates the problem, this still weakens its applicability in real world scenarios [2]. Alongside, as all nodes belonging to the same group are fully-replicated, the available storage space provided by the group is limited to the size manageable by the single node with the lesser storage capabilities. This restriction is of special importance if we consider each node to be commodity hardware or even smaller edge devices where storage space available is limited.

In order to tackle both limitations just described, we propose the integration of DATAFLASKS with data deduplication, a widely used technique for eliminating duplicate data efficiently in storage systems, that has also proven to be useful for reducing network bandwidth usage [25, 24]. Briefly, by avoiding the storage of duplicate content, extra space is available for keeping additional data. Similarly, duplicate content can be detected before being sent through the network, thus sparing network bandwidth usage. As shown in the paper, deduplication mechanisms allow significant storage savings in each node, and allow improving the pro-active replication mechanism and its network bandwidth consumption. The result is a massive scale data store that is space efficient and that effectively reduces bandwidth consumption while maintaining its highly desirable resilience characteristics.

Along this paper, we describe how deduplication techniques are integrated into DATAFLASKS ultimately building DDFLASKS, a massive scale deduplicated data store. Additionally, we show the applicability of DDFLASKS and its efficacy using a real world application. In particular, we use this novel system to serve as the storage infrastructure for Wikipedia [13]. Our system allows storing and serving simultaneously both the most recent versions of Wikipedia articles as well as older historical versions of the same articles. Moreover, articles are judiciously stored to maximize the effectiveness of the deduplication mechanisms. In fact, using real data dumps from Wikipedia, we show that our system is able to store and serve articles across several nodes with high levels of storage savings (from 45% up to 63%) and network savings (up to 20%).

The rest of the paper is organized as follows. In Section 2 we describe the architecture and structure of DATAFLASKS, the baseline system used to build our novel approach. Next, in Section 3 we describe the Wikipedia use case and present some preliminary results that motivate the usage of deduplication. In Section 4 we introduce DDFLASKS. We then proceed to DDFLASKS evaluation in Section 5 and present related work in Section 6. The paper is concluded in Section 7.

2 DATAFLASKS: Scalable storage

The pivotal idea guiding the design of DATAFLASKS is decentralization [22]. In DATAFLASKS each node is autonomous and all nodes play the same role. A node progresses relying solely on local decisions without depending on any other node and on any kind of hierarchy. When a client issues a request, such request is disseminated throughout the system and each node decides how to handle it. Store requests are composed by an identifier of the object to be stored that must be unique, by the version of the object to be stored, and by the object itself (the actual data to be stored). Storing several versions of the same object is important for many applications that resort to data versioning.

Briefly, the API is composed by a *get* and *put* operation. When a *get* is received, if the node holds the corresponding triple (key,version,object) it replies to the client. Otherwise, it ignores the request. In the case of a *put* operation, the node locally decides to store the corresponding triple (key,version,object) or to discard it. The decision to store or not the data is used to implement data distribution and replication. DATAFLASKS is designed in such a way that prevents all nodes to take the same decisions, which would lead to a system where all nodes either store every object or none at all. Both situations are undesirable as the former prevents data distribution and the latter defeats the purpose of the data store. At the same time, DATAFLASKS ensures that a sufficient number of nodes actually decides to store each data object in order to guarantee data replication, and consequently, to tolerate node failures.

The set of nodes that takes the same decisions on whether to store data objects or not is viewed as a group. Accordingly, the decision of which data to store is reduced

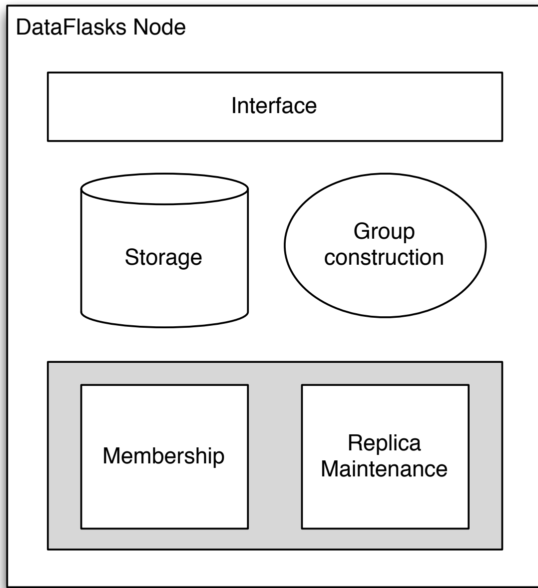

Figure 1: DATAFLASKSnode high level architecture.

to the decision of which group a node belongs to. Once that decision is made, each node is responsible for a subset of the data according to a deterministic mapping between the pair (key,version) of an object and the group it belongs to. Data is thus distributed by groups, providing load balancing, and replicated a number of times equal to the size of the group. Strikingly, each node is able to decide to which group it belongs without requiring any kind of coordination.

In order to achieve this, the system is entirely built with epidemic protocols. In particular, unstructured and pro-active epidemic protocols. They are characterized by their independence from any kind of structure or hierarchy among nodes and by the fact that they rely on pro-active mechanisms for fault tolerance. Instead of explicitly detecting failures and act accordingly, pro-active protocols are continuously taking the initiative, being able to anticipate system repair. The result is a completely decentralized and coordination-free data store. Characteristics that make DATAFLASKS inherently scalable and able to cope with unprecedented levels of system dynamism, may it be caused by membership instability or by failures.

The architecture of a system node is depicted in Figure 1. Each node runs five components: Membership, Group Construction, Storage, Replica Maintenance and Interface. In order to provide some background and context to the design of the system proposed in this paper, we briefly describe how each component works in the original setting.

Membership. This component is responsible for providing each node with a list of available nodes in the system. It does so guaranteeing that such list represents a random sample of nodes from the entire system and that it is periodically refreshed. This component relies on Cyclon [31], a Peer Sampling Service [18], to achieve this. It is important to notice that each membership list is always a small subset of nodes with respect to the system size, which allows the system to scale. However, if such size is carefully chosen, the resulting set of views yields an overlay network that allows for reliable data dissemination with very high probability [10, 11]. This is the component that supports communication between nodes.

Group Construction. This component is responsible for determining to which group the node belongs. As described previously the group determines which data to locally store or to discard. Without going into much detail, this component works by leveraging information being propagated at the membership level to estimate the number of groups needed to satisfy a desired, user defined, replication factor. Then, the node places himself on one of those groups guaranteeing that system nodes are uniformly distributed across the different groups. For a detailed description of the protocol please refer to [22]. Once in a group, each time a *put* operation is issued for a certain key, that key is mapped deterministically to a group by using an hash function. As described further on, this mapping allows different versions of the same key to be placed in the same replication group. This will allow maximizing the deduplication mechanism effectiveness.

Storage. The storage component abstracts the actual medium to which the data is persisted. Currently, this component can be configured to be a in-memory store or a disk-based one. In this paper we design and implement a new storage component to allow data deduplication.

Replica Maintenance. Within a group, all nodes store the same set of data objects. In order to maintain the replication level in the presence of churn, the replica maintenance component periodically publishes to other nodes in the group the set of keys it currently holds locally. Upon receiving a maintenance message, each node checks if it is storing all keys correspondent to the group. If not requests the missing data from the nodes in its group. In this paper we provide a new replica maintenance component which allows to optimize this process by avoiding to transmit duplicate data through the network.

Interface. Finally, the interface component is responsible for handling the incoming connections from other nodes and managing the request workflow in the system.

In order to issue *put* or *get* requests the client only needs to be able to contact a single node in the system. The request is then forward appropriately to the correct nodes that can fulfill it. By knowing a set of nodes in the system it is then possible to implement different rooting strategies for incoming client requests.

3 Duplicates in the real world

Many large information systems tend to exhibit a significant amount of duplicate data [23]. This is particularly true for storage systems that evolve incrementally with time. A paradigmatic example is Wikipedia, also known as the Internet encyclopedia [13]. The Wikipedia allows users to create, edit or complement articles about virtually any subject. Each article can evolve through time and periodic snapshots of the entire Wikipedia database are stored for future reference [12]. Because Wikipedia data serves a very high volume of requests (it is considered to be among the ten most popular websites[1]) and stores a large volume of data, that is constantly evolving over time, it is a suitable use case for DATAFLASKS that can leverage its highly scalable infrastructure to serve Wikipedia's high demand.

Naturally, different versions of the same article share significant portions of the text, which is redundant when stored. This means that a storage system holding the full history of Wikipedia is expected to have a considerable amount of duplicate content [14]. One possible approach to eliminate such redundancy and to spare storage space is to use incremental backup techniques such as delta-encoding. With this technique new versions of a previously stored article are stored as deltas or *diffs* that only contain the content that was actually modified. These deltas can then be applied to the original (base) article to rebuild a specific version of the article. Although this technique is efficient in terms of storage space savings, it requires additional computational power and it is slower than deduplication, specially when articles have a large number of versions and several deltas must be applied to the base article to retrieve latest versions [25]. For this reason, this paper proposes the use of block-based deduplication, which allows users to query any article version in the past and get the response without the need to rebuild a set of deltas. Further details on the deduplication mechanisms implemented are discussed in Section 4.

In order to validate that deduplication is, in fact, suitable and effective for a deployment where DATAFLASKS is serving Wikipedia articles, we performed the following experiment. We used 15 monthly Wikipedia snapshots taken for the period between November of 2014 and January of 2016 [12]. Each snapshot has the latest full version of all articles belonging to the English version of Wikipedia. The snapshots were processed by the order they were taken and the corresponding articles were stored in a way that mimics the distributed storage approach taken by DATAFLASKS in a real deployment i.e., articles were divided into groups and stored accordingly. Each group of articles represents the data partition that would be assigned to a DATAFLASKS node. We then focus our analysis on each one of the partitions. It is important to notice that deduplication will be applied locally by each node. Consequently, nodes that replicate the same data partition will behave similarly, which makes it sufficient to analyze a single node behavior per group, i.e. the behavior of the storage for each group of articles. Additionally, across consecutive snapshots there are some repeated articles that remained unmodified so these articles were not stored in our experiment. On the other hand, new versions of previously stored articles were routed to the same data group where their ancestors were kept and are stored as new objects (files) with distinct version ideintifiers. This way, the experiment stores the full content for each article version which is in conformity with the rationale explained previously where our very large data store is used to serve several articles and their distinct versions without requiring the usage of incremental backup techniques.

After populating the distinct data groups with the Wikipedia dataset the global storage space in use was $\approx$ 305 GB, corresponding to 55,745,648 articles. In order to check the percentage of redundancy in the stored dataset, we resort to the DUPSANALYSER tool[2] an open-source project that processes the content of files and extracts statistics for the duplicate content found. Duplicates can be found either by searching for duplicate blocks with a fixed or variable size.

The latter resorts to an implementation of the Rabin Fingerprint scheme for calculating variable-sized blocks and their corresponding content hashes efficiently [24]. As Wikipedia articles are text articles, using variable sized blocks is a better choice for finding duplicates [25, 14]. Briefly, lets consider two versions of the same article where version A only differs from version B by a single character that was added to the beginning of the latter version. If the two articles are scanned with a fixed size partitioning scheme, no blocks from version A will match blocks from version B. In contrast, the Rabin fingerprint scheme uses a sliding window that moves through the data until a fixed content pattern defining the block boundary is found. This approach generates variable-sized blocks and solves the issue of inserting a single byte in the beginning of version B. More precisely, only the first block from version B will differ from the first block of version

[1] http://www.alexa.com/siteinfo/wikipedia.org

[2] https://github.com/jtpaulo/dupsanalyzer

Table 1: Average and standard deviation for the percentage of duplicates found per group with 1024, 2048 and 4096 bytes Rabin fingerprints for DataFlasks configurations with 40, 20 and 10 groups.

# groups	1024 bytes	2048 bytes	4096 bytes
10	42.95 ($\pm$ 0.11) %	33.97 ($\pm$ 0.13) %	24.63 ($\pm$ 0.13) %
20	42.84 ($\pm$ 0.17) %	33.90 ($\pm$ 0.18) %	24.60 ($\pm$ 0.20) %
40	42.74 ($\pm$ 0.24) %	33.84 ($\pm$ 0.26) %	24.57 ($\pm$ 0.28) %

Table 2: Analysis of duplicates results with 1024, 2048 and 4096 bytes Rabin fingerprints for a single group of the DataFlasks configuration with 40 groups.

Fingerprint Avg Size	# articles	Total space (GB)	total # blocks	# unique blocks	# duplicate blocks	Avg # copies / duplicated block	Space saved (GB)	% duplicate space
1024	1,393,130	7.63	7,046,744	4,226,205	2,820,539	3.20	3.27	42.88
2048	1,393,130	7.63	3,995,416	2,870,780	1,124,636	2.59	2.59	33.99
4096	1,393,130	7.63	2,550,938	2,132,849	418,089	2.65	1.89	24.81

A due to the byte addition, while the remaining blocks will still be duplicate. Finally, the Rabin scheme is configurable with target average, maximum and minimum block size, which allows avoiding the generation of very small or large blocks while still keeping their sizes variable. In the results discussed next, we used DUPSANALYSER to process the articles stored at each data group, that were stored as independent files[3], in order to analyze the redundancy found in a per-node basis. Individually, for each data group, our analysis tool processed each stored file to find intra and inter file duplicates.

Distinct group sizes results Our first results show the amount of duplicates found per group when dividing articles into different number of groups, namely, 10, 20 and 40 groups. Intuitively, when the number of groups is larger, each group holds less data (articles) from the original dataset. In this example, with 10 groups each group holds $\approx$ 30GB, with 20 groups $\approx$ 15GB and with 40 groups $\approx$ 7.5GB. As shown in Table 1 the percentage of duplicates found do not increase significantly if a group holds more data. This happens because most redundancy is originated by storing distinct versions of the same article in the same group, which happens identically for the 3 group sizes. In the table it is also visible that for larger bock sizes, the percentage of duplicates is reduced. With a larger block size, small portions of the blocks that are redundant are not eliminated, which explains this result. The block sizes specified in the table are the average sizes defined for the Rabin Fingerprinting scheme. The maximum and minimum sizes defined are calculated by adding/subtracting $averagesize/2$ to the $averagesize$.

[3]Each article version was also stored as an independent file with the full content for that version

For the blocks with an average size of 1024 bytes, it is possible to save almost 45% of the occupied storage space per group. At the same time, the standard deviation values are very small meaning that the percentage of duplicates found in distinct groups is very similar. To sum up, these results show that, potentially, 45% of the english Wikipedia storage space can be saved with deduplication with 1K variable sized blocks.

Single group analysis for the 40 groups scenario Since the percentage of duplicates does not change significantly when considering different number of groups, we show in Table 2 a more detailed analysis of the stored content in a single group for the experiment with 40 groups. The analyzed group holds 7.63 GB of data corresponding to more than one million articles. For each Rabin fingerprint size the total number of generated blocks diverges and, as expected, with a smaller size it is possible to find more duplicates and have significantly higher space savings. On the other hand, reducing the block size increases the size of the metadata used to index all stored blocks and to find duplicates. As shown in Section 5, even with an average block size of 1024 bytes it is possible to save both storage and network consumption while having an acceptable metadata size in each group.

As shown in Table 3, generated blocks have an average size near to the expected one. However, since each article is processed and partitioned independently, there is a small chance that the last block for the article ends up with a size that is inferior to the minimum value set for the Rabin fingerprinting scheme. As shown in the same table the number and the storage space occupied by these blocks are minimal, specially when the fingerprint average size decreases. Nevertheless these incomplete blocks

Table 3: Analysis of complete and incomplete blocks with 1024, 2048 and 4096 bytes Rabin fingerprints for a single group of the DataFlasks configuration with 40 groups.

Fingerprint Avg Size	# incomplete blocks	incomplete blocks space (MB)	Avg size / incomplete block (B)	# Complete blocks	Complete blocks space (GB)	Avg size / complete block (B)
1024	524,215	164.00	328.04	6,522,529	7.47	1229.13
2048	812,976	433.73	559.43	3,182,440	7.20	2430.27
4096	958,372	794.70	869.50	1,592,566	6.85	4618.77

are also accounted in Section 5 in terms of storage and metadata space required.

To conclude, these results show that single-group deduplication with a variable 1KB fingerprinting scheme allows reducing $approx$ 45% of the storage space occupied by 15 snapshots of the English Wikipedia version. Note that these snapshots only correspond to a 1 year and 3 months period and as the number of snapshots increases the deduplication space savings will be even higher. This motivates the next sections where we show how deduplication can be applied to DataFlasks and, additionally, show that deduplication is not only useful for reducing storage usage but also network bandwidth consumption.

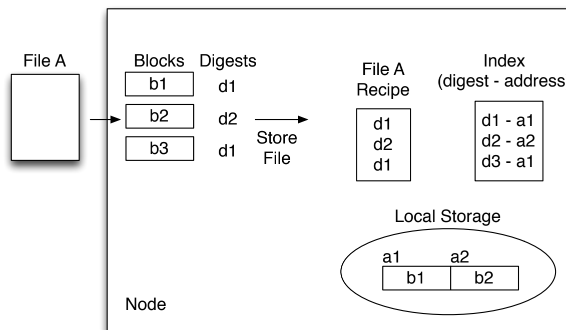

Figure 2: Deduplication in DDFlasks

4 DDFlasks

Recalling Section 2, data distribution and replication in DataFlasks is achieved by dividing nodes into groups. Each group is responsible for a set of data and, accordingly, each node belonging to that group will have to store that specific set of data in its local storage. The Wikipedia study discussed in the previous section shows that a significant percentage of duplicates exists in each node when all the versions of a specific article are grouped together. In DDFlasks, this insight is leveraged by ensuring that data objects (articles) identified by a key are always assigned to the same group independently of their version. With this approach, all the versions of an article are stored in the same group while clients can still retrieve specific versions of an article by specifying the article's key and the desired version. This is achieved by taking into advantage the load balancing mechanism from the original DataFlasks, which deterministically routes a certain key to a group. As a consequence, different versions for the same key are routed to the same group. This decision is key for obtaining significant storage spacings while performing node-local deduplication. The advantage of local deduplication is that it does not require any global index or coordination mechanisms that would impact further the performance of storage requests [25]. This way, DDFlasks design remains fully-decentralized *i.e.*, nodes progress solely based on local decisions and without the need for any structure or hierarchy between them.

Architecture In comparison with the baseline architecture discussed in Section 2, DDFlasks is extended with two additional deduplication mechanisms. The resulting system is available as open source software at `http://github.com/fmaia/dataflasks`.

First, a new storage component is provided with integrated in-line local storage deduplication, which works as follows. In each node, duplicates are identified and eliminated before actually being stored persistently. In the literature this approach is known as in-line deduplication [25]. Duplicates are found by resorting to an *index* that maps blocks with unique content to their respective storage addresses. When a block is being written a digest of the block's content is calculated and the index is searched for a possible duplicate. If a duplicate exists, then the new block does not need to be stored, otherwise, the block is stored and the index is updated with a new entry for that block. A Rabin Fingerprint scheme identical to the one described in Section 3 is used to divide files into variable-size blocks and to calculate small digests of their content [24]. This way, the index does not store the actual content of the block but a smaller digest identifying the content of that block. In order to retrieve files from the storage system, an additional metadata structure, that we refer to as *file recipe* is used. Each file recipe identifies

a single file stored on DDFLASKS and tracks the digests of the blocks that belong to that specific file. The actual storage address of these digests can be consulted at the index. Deduplication is thus achieved because file recipes with duplicate content share digests that are mapped to the same storage block. Figure 2 shows an example of the proposed single-node deduplication mechanism. As the first step, File A is routed to the correct group of nodes. Then, in each node storing the file, the file is divided into variable-sized blocks and a digest for the content of each block is calculated. In the example, block1 and block3 have the same content. Each digest is checked at the index and if not found, a new entry is added while the corresponding block is stored in a append-only storage. In the figure blocks $b1$ and $b3$ are duplicates, so only block $b1$ and $b2$ are stored. Finally, the file recipe for File A is also kept at the node in order to fetch all the necessary blocks when a client asks for that file. The index keeps the digests and corresponding location for all blocks at the local storage which enables both intra- and inter-file deduplication for all files stored in the same node. In Section 5 we show that our approach is still able to achieve significant storage space savings even when metadata space is accounted for.

In the context of the current paper we do not address data deletion functionalities. This is motivated by the fact that DDFLASKS is a large-scale system intended to store large amounts of archival data. For use-cases such as the Wikipedia one used in the paper, this is a practical assumption since the main goal is to keep all versions of wikipedia articles without ever deleting them. Moreover, because we are targeting very large-scale epidemic storage systems, we follow an in-line approach similar to the ones proposed in previous work [26, 8]. This allows our proposal to avoid scalability issues found in large-scale in-line deduplication systems that must maintain a global index for all storage nodes [8, 7].

The second deduplication mechanism proposed in the paper aims at optimizing the network bandwidth used by DDFLASKS data replication techniques. In order to cope with high levels of node churn and to maintain desirable data replication levels, each system node proactively and periodically contacts other nodes in the same group to announce the set of files it is currently storing. If one node receives this set and verifies that its local storage is currently missing some files, it must contact other nodes in the same group to ask for those files. Naturally, when churn levels become significantly high, the volume of data traversing the network increases as more files are being exchanged. We propose to mitigate this problem by employing deduplication to the data being exchanged between nodes. In detail, nodes periodically announce to the group not only the set of files that they currently hold but also the digests that compose those files. When a node receives this list and verifies that a set of files is missing, it checks first what digests from those files are already stored locally. This can be done by leveraging the index metadata used for local storage deduplication. Then, the node only requests from the other group nodes the blocks that are actually missing in its local storage. After receiving these blocks the node updates the index and creates the corresponding file recipes. With this approach, only missing blocks are sent through the network and not the whole files. A key advantage of this mechanism is that it relies on the metadata already used for performing in-line deduplication, which is an idea that has proven successful in previous proposals for backing up data across peer-to-peer networks [24, 5]. Although this strategy requires sending the list of digests when announcing the files that nodes currently hold, we show in Section 5 that it still spares significant network bandwidth.

Implementation details The two deduplication mechanisms were implemented on top of the current implementation of the system described in Section 2. The deduplication index is implemented as an in-memory HashMap that maps blocks digests (8 bytes) to storage addresses (8 bytes)[4]. Similarly, file recipes are stored in an in-memory HashMap that maps the identifier of a file (16 bytes, 8 bytes for the file key and 8 bytes for the version) to its file recipe whose size depends on the number of block digests composing that file. DDFLASKS is mainly thought for running in commodity hardware nodes and the amount of data hold by each node is not expected to be very large (tens to hundreds of GBs). This means that the amount of metadata held by each node is also expected not grow to large values. The next section gives further details for the size of metadata expected for a specific storage size. Additionally, in the context of this paper we assume that, even in the presence of high levels of churn, for each group there is always a set of live nodes. This means that metadata for freshly booted nodes can always be reconstructed from live nodes. Moreover, the index and file recipe are periodically stored on disk to ensure that when a node is rebooted, some of the previously stored metadata is already in the node and does not need to be requested from other nodes.

The inter-node communication in DDFLASKS is achieved by relying in the UDP protocol. The connectionless characteristics of this network protocol withdraws the need to maintain connections, however it introduces unreliability in the delivery of messages. In order to minimize package loss, DDFLASKS splits message payloads that exceed UDP's maximum transfer capacity into as many parcels as required, reuniting them upon reception.

[4]For each entry at the index, 4 extra bytes must be stored because variable sized blocks are being used and their size must also be kept.

Notwithstanding the fact that such mechanism allows delivering payloads bigger than what is allowed by a single UDP datagram, it does not prevent package loss. To mitigate this possibility, DDFLASKS ensures that packages are re-transmitted after a configurable time threshold, thus ensuring the delivery of messages.

5 Evaluation

DDFLASKS was evaluated in a real deployment to validate two main claims. First, that deduplication allows sparing significant storage space for each node. Second, that the network bandwidth used by nodes when exchanging messages is also reduced.

To this end, we have performed a set of experiments that demonstrate the effectiveness of the deduplication mechanism implemented. Each experiment was run both in the original DATAFLASKS, non-deduplicated system (used as the baseline) and in DDFLASKS. The experiment set up consists of a cluster of commodity hardware nodes equipped either with a 3.1 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk or a 3.7 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a SSD disk. All nodes are connected through a gigabit ethernet switch. It is important to notice that hardware heterogeneity does not impact the results of our experiments. In fact, it is out of the scope of the present paper the evaluation of system performance metrics. Instead, we focus on analyzing storage and network savings achievable by our system. Similarly, the validation of DDFLASKS scalability to thousands of nodes and resiliency to high churn rations is already addressed in previous work [22].

Leveraging the results obtained in Section 3 and aiming at real world assessment of DDFLASKS, all the experiments presented next resort to actual Wikipedia data.

5.1 Storage Savings

In order to evaluate the storage behavior of DDFLASKS we have considered 15 Wikipedia monthly snapshots. Each one of these snapshots contains a set of articles from the English version of the Wikipedia. From snapshot to snapshot each article may change reflecting its evolution through time. In the real world deployment of Wikipedia, users see only a single (latest) snapshot. However, in our scenario we want to go a step forward and it is our goal to simultaneously store and serve several Wikipedia snapshots.

The 15 snapshots used amount to $\approx$ 115 GB corresponding to $\approx$ 6.3 million articles. Each article is stored as a single data object in the storage system and each new article snapshot corresponds to a new version of such object. Moreover, article versions are treated as new articles but are identified with the same key as the original article and different version number. This information is used by DDFLASKS to collocate articles with their subsequent versions.

We configured both DATAFLASKS and DDFLASKS to arrange nodes into 16 groups. Each group is responsible for storing a subset of the articles written to the store. As described previously, all nodes belonging to a certain group store the same data and deduplication is applied local to each node. Consequently, in order to observe the system's behavior it is sufficient to analyze the behavior of a single node per group. Other nodes in the same group will exhibit exactly the same results.

The experiment consisted on loading both DATAFLASKS and DDFLASKS with the 15 data snapshots writing each article and subsequent versions in chronological order (from the oldest snapshot to the latest one). After the load was completed we analyzed the storage usage of a node per group.

In Table 4 we present the results of this experiment. It is observable that DDFLASKS is significantly more frugal than DATAFLASKS with respect to storage space usage. The former requires 42.4 GB to store all the articles while the latter, without deduplication, requires 115.5 GB. In detail, 73.1 GB are saved by using deduplication which corresponds to a space saving of 63% when compared to the baseline approach. Please note that, when compared with the motivation tests described in Section 3, there is an improvement in the storage savings results. This improvement is explained by the fact that, in this real deployment, we used a sample of the articles (and corresponding versions) used in the motivation experiments, which happen to exhibit slightly higher redundancy between them. Nevertheless, this experiment validates the effectiveness of deduplication with respect to storage consumption. Additionally, we can observe that the local storage space required by nodes in different groups is similar and that the deduplication savings in each node are identical to the one observed globally for the whole storage. This is a direct consequence of using a load balancing strategy that routes articles uniformly across distinct groups.

Going into some detail, we also show in the table the space used by metadata structures. In both systems, more than 390,000 articles were stored in each node. As expected, deduplication requires additional metadata space for storing and indexing articles' blocks, while in the baseline system it is only required a simpler file recipe that points a specific file to its storage address. Nevertheless, the space savings achieved clearly compensate the overhead introduced by the extra metadata structures used in DDFLASKS. In fact, less than 17% of the space spared by deduplication is needed for fulfilling the extra metadata space overhead. Finally, Table 5 shows the exact space occupied by the index and file recipe metadata in our sys-

Table 4: Storage and metadata space occupied for DDFLASKS and the DATAFLASKS storage systems

	DATAFLASKS	DDFLASKS
Global storage space (GB)	115.5	42.4
Average storage space / node (GB)	7.2 ($\pm$ 0.08)	2.65 ($\pm$ 0.05)
Global Deduplication savings (GB)	-	73.1
Average deduplication Savings / node (GB)	-	4.55
Global Metadata space (GB)	1.32	12.04
Metadata space / node (GB)	0.08 ($\pm$ 0.003)	0.75 ($\pm$ 0.05)

Table 5: Space occupied by DDFLASKS index and file recipe

Metadata	Global space (GB)	Space / node (GB)
Index	5.35	0.33 ($\pm$ 0.002)
File recipe	6.69	0.42 ($\pm$ 0.003)

tem. Again, the space occupied by each metadata structure across different nodes does not change significantly.

5.2 Network Savings

Replication is achieved in our system resorting to periodic message exchanges between nodes with information about the data objects they are storing. Each time, following a message exchange, a node detects it is missing some object it requests it from other nodes in the same group. Naturally, if the system is stable, it is expected that nodes store all correspondent data objects and that these message exchanges do not yield missing data requests. However, when nodes fail or enter the system data objects need to be requested to maintain the desirable replication levels.

In this experiment, we show that deduplication can reduce network consumption of the data exchange mechanism between nodes. We focus on two nodes belonging to the same group and observe their behavior when one of them keeps failing and re-entering the system while the system is continuously being loaded with new data. Naturally, it is expected that each time the node re-enters the system it will request missing data from its peer that runs continuously.

The test ran for 2 hours and after the first 30 minutes one of the nodes was stopped in intervals of 20 minutes. In detail, after being stopped the node remains offline for 20 minutes and then it is rebooted again and it is kept online for additional 20 minutes. This cycle was repeated until the last 30 minutes of the test when the two nodes were kept online. The node being stopped saved its metadata to disk periodically to ensure that when rebooted the index and file recipe metadata were holding previously stored information.

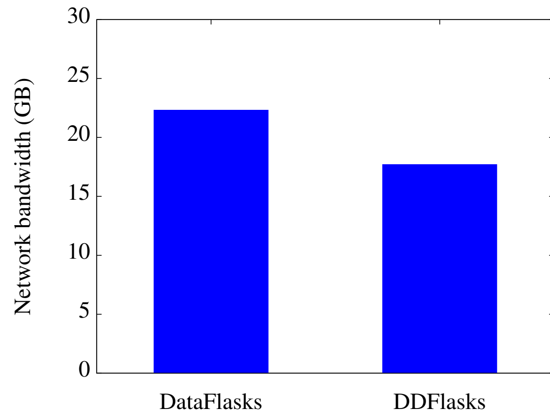

Figure 3: Network consumption in DATAFLASKS and DDFLASKS

Again, 15 Wikipedia monthly snapshots were used, and both systems (DDFLASKS and baseline) stored more than 400,000 articles, which corresponds to ≈8.3GB. Please recall that the two nodes were configured to be in the same group so these were fully-replicated, each holding the same amount of articles mentioned previously. In terms of storage space savings the DDFLASKS nodes stored 4.3 GB while the baseline system nodes stored 8.3 GB. This corresponds to a space saving of ≈49%, which is in conformity with the results discussed previously and in Section 3. The metadata space required by each node is also compensated by the space savings as in the previous results.

Figure 3 shows the network consumption for DDFLASKS and the baseline approach without network deduplication. The results show that the baseline approach sends more than 22 GB through the network while the deduplication approach only sends 17.71 GB. Note that these bandwidth consumption results consider all network traffic. In fact, while most of this traffic is due to the data replication mechanism, system control traffic and client requests are also accounted for in the total

value. Moreover, as discussed previously, both systems rely on the UDP protocol that requires resending messages that are lost due to failures of the protocol, which also increases network bandwidth usage. Nevertheless, these results show that only by using deduplication for the data replication mechanism it is possible to spare ≈20% of all the data exchanged across replicated peers.

The results described in this section support our claim that by using deduplication in DDFLASKS it is possible to spare significant storage (more than 49%, depending on the Wikipedia sample used) and network usage (20%).

6 Related work

In the pursuit for large scale data management, traditional relational database systems have been, for certain domains and applications, largely replaced by new approaches to data management. Commonly know as NoSQL data stores, these data management systems offer relaxed consistency guarantees when compared with traditional relational database management systems. Examples are Dynamo, PNuts, Bigtable, Cassandra and Riak [6, 4, 3, 20, 19]. One of the key features of these data stores is how they implement data distribution and discovery. Leveraging scalability properties of peer-to-peer protocols, all these data stores rely on a distributed hash table (DHT) such as Chord or variants to distribute and locate data objects [29]. The exceptions are Bigtable and PNUTS, which are centrally managed instead. However, it is important to notice that these data stores typically use a specific DHT variation called 'one-hop' DHT [16, 30]. This variation allows faster lookups but requires complete membership knowledge, i.e., each node knows about all other nodes in the system. Moreover, DHTs are know to struggle in the presence of high levels of churn [27]. As a result, even if the distributed and peer-to-peer nature of these data stores is closely related to DATAFLASKS, this system presents an unique unstructured and pro-active approach to node organization and data replication.

To our best knowledge, applying deduplication to epidemic massive scale systems such as DATAFLASKSfor improving the usable storage space of peers and to improve the network bandwidth usage of gossip protocols and pro-active replication mechanisms is a novel contribution of this paper. To achieve these goals, this paper leverages ideas of previous work on deduplication for distributed storage systems [25]. In more detail, for achieving both storage and network savings, in-line deduplication is applied so that duplicates are eliminated before being stored persistently [26, 8]. In fact, for sparing network bandwidth, duplicates are eliminated before even being sent through the network [24].

Peer-to-peer in-line deduplication, where backups are made cooperatively with remote nodes, was introduced in Pastiche [5]. In this system, nodes backup their data to other remote nodes that are chosen by their network proximity and data similarity. Only non-duplicate data is sent through the network and since nodes with similar datasets are chosen, the amount of data that must be sent through the network and stored in each peer is reduced significantly. Other distributed deduplication systems propose novel load balancing designs that route similar data to the same node in order to optimize the amount of duplicates found and, consequently, maximize storage space savings. These proposals rely on centralized indexes that have global knowledge of the content stored in all nodes, on distributed indexes that scale better than the centralized ones, on statefull and stateless routing algorithms, and on probabilistic routing algorithms that do not need a global knowledge of the content of each node in the system [8, 9, 21, 1, 7, 14, 32, 15].

Although DDFLASKS could benefit from some of the ideas and optimizations discussed in previous deduplication systems, our current design uses the original load balancing algorithm proposed by DATAFLASKS. Our approach collocates different versions of the same data objects, which are expected to have duplicated content. Deduplication is thus performed locally on each node *i.e.*, each node manages its own index and only eliminates duplicates that are stored on its local storage. Strikingly, as shown in the paper, for realistic use-cases such as the Wikipedia one, ensuring that the same versions of articles are routed to the same DDFLASKS group is enough to achieve significant storage space savings while keeping metadata overhead acceptable. Additionally, our deduplication design can be leveraged to spare not only storage space but also network bandwidth usage across nodes. For epidemic data stores such as DDFLASKS this is, to our best knowledge, a novel contribution that reduces significantly the number of messages exchanged across nodes, thus improving the efficiency of current gossip protocols, which is of particular importance since bandwidth consumption is critical in these systems [2].

7 Conclusion

In this paper we propose a deduplicated very large scale data store. Its main goal is to handle massive scale amounts of data minimizing storage resource usage while being highly scalable and resilient to node churn. To achieve this goal, DDFLASKS is built resorting to a stack of proactive and completely decentralized gossip-based protocols. The core idea driving this store is effective data dissemination and independent, local decisions of what to do with the data at each node. In-line deduplication is employed at each node and we show, resorting to a real world

scenario, that the system is able to save up from 45% up to 63% of storage space.

DDFLASKS is able to cope with unprecedented amounts of churn at the cost of constant message exchanges between nodes. Naturally, this results in high bandwidth consumption. Interestingly, we show in this paper that deduplication mechanisms introduced to save storage space can also be used to reduce the amount of data sent through the network. Our evaluation shows savings of up to 20% in network bandwidth consumption.

By introducing local deduplication at each node our approach does not influence the scalability and resilience of the data store as a whole. DDFLASKS can still adapt to deployments of thousands of nodes and endure very high churn rates. In more detail, local deduplication does not require any kind of distributed coordination mechanism neither impacts the replication mechanism. Nodes still progress solely based on local decisions and without the need for any structure or hierarchy between them. Moreover, not requiring a global knowledge of the content stored across all nodes also helps mitigating the overhead known to be induced by current distributed in-line deduplication proposals.

Considering scenarios such as data backup, versioned data storage or data archival, the use of in-line deduplication proves to be very effective, even when performed locally. Not only storage savings are significant but, additionally, from the point of view of the client, data becomes available and accessible independently of its age or version. Taking as an example the Wikipedia, our use case throughout the paper, DDFLASKS is able to serve different, historical versions of Wikipedia articles with reduced computational impact and with very significant storage savings.

The combination of deduplication and very large scale data management is, to the best of our knowledge, a novel contribution of this paper. It renders DDFLASKS a suitable system for massive scale data storage and management.

References

[1] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. In *Proceedings of International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–9. IEEE, 2009.

[2] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[3] 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)*, 26(2):4, 2008.

[4] 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*, 1(2):1277–1288, 2008.

[5] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–13. USENIX, 2002.

[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[7] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in Scalable Data Routing for Deduplication Clusters. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 15–29. USENIX, 2011.

[8] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming Space from Duplicate Files in a Serverless Distributed File System. Technical Report MSR-TR-2002-30, Microsoft Research, July 2002.

[9] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a Scalable Secondary Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 197–210. USENIX, 2009.

[10] P. Erdős and A. Rényi. On the evolution of random graphs. *Selected Papers of Alfréd Rényi, vol*, 2:482–525, 1976.

[11] P. Eugster, R. Guerraoui, A. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *Computer*, 2004.

[12] W. Fountation. Wikimedia downloads web page. `https://dumps.wikimedia.org`, 2016.

[13] W. Fountation. Wikipedia web page. `https://www.wikipedia.org`, 2016.

[14] D. Frey, A.-M. Kermarrec, and K. Kloudas. Probabilistic deduplication for cluster-based storage systems. In *Proceedings of the Third ACM Symposium on Cloud Computing (SOCC)*, pages 1–14. ACM, 2012.

[15] Y. Fu, H. Jiang, and N. Xiao. A Scalable Inline Cluster Deduplication Framework for Big Data Protection. In *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, pages 354–373. ACM, 2012.

[16] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. *NSDI*, 2004.

[17] IDC. The digital universe of opportunities: Rich data and the increasing value of the internet of things, April 2014.

[18] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.

[19] R. Klophaus. Riak core: building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[20] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[21] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 111–123. USENIX, 2009.

[22] F. Maia, M. Matos, R. Vilaa, J. Pereira, R. Oliveira, and E. Rivire. Dataflasks: Epidemic store for massive scale systems. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, pages 79–88, Oct 2014.

[23] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage*, 7(4), 2012.

[24] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proceedings of Symposium on Operating Systems Principles (SOSP)*, pages 174–187. ACM, 2001.

[25] J. Paulo and J. Pereira. A Survey and Classification of Storage Deduplication Systems. *ACM Computing Surveys*, 47(1):11:1–11:30, 2014.

[26] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, pages 1–13. USENIX, 2002.

[27] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX, 2004.

[28] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*. USENIX, 2007.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, 2003.

[30] R. Vilaça, F. Cruz, and R. Oliveira. On the expressiveness and trade-offs of large scale tuple stores. *On the Move to Meaningful Internet Systems, OTM 2010*, pages 727–744, 2010.

[31] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[32] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *Proceedings of USENIX Annual Technical Conference (ATC)*, pages 26–30. USENIX, 2011.

[33] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2008.