Týr: Efficient Transactional Storage for Data-Intensive Applications

Pierre Matri, Alexandru Costan, Gabriel Antoniu, Jesús Montes, María S. Pérez

Týr: Efficient Transactional Storage for Data-Intensive Applications

Pierre Matri, Alexandru Costan, Gabriel Antoniu, Jesús Montes, María S. Pérez

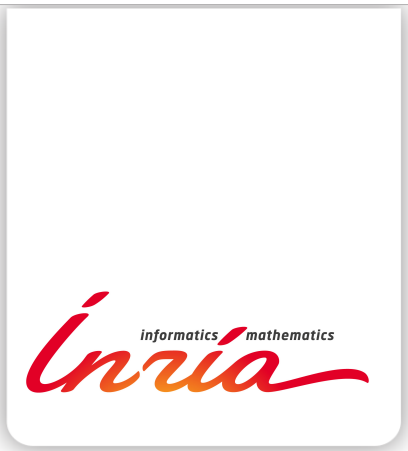
TECHNICAL REPORT

N° 473

January 2016

Project-Team KerData

ISSN 0249-0803 ISRN INRIA/RT--473--FR+ENG



Týr: Efficient Transactional Storage for Data-Intensive Applications

Pierre Matri*, Alexandru Costan†, Gabriel Antoniu‡, Jesús Montes*, María S. Pérez*

Project-Team KerData

Technical Report n° 473 — January 2016 — 25 pages

Abstract: As the computational power used by large-scale applications increases, the amount of data they need to manipulate tends to increase as well. A wide range of such applications requires robust and flexible storage support for atomic, durable and concurrent transactions. Historically, databases have provided the *de facto* solution to transactional data management, but they have forced applications to drop control over data layout and access mechanisms, while remaining unable to meet the scale requirements of Big Data. More recently, key-value stores have been introduced to address these issues. However, this solution does not provide transactions, or only restricted transaction support, compelling users to carefully coordinate access to data in order to avoid race conditions, partial writes, overwrites, and other hard problems that cause erratic behaviour. We argue there is a gap between existing storage solutions and application requirements that limits the design of transaction-oriented data-intensive applications. In this paper we introduce Týr, a massively parallel distributed transactional blob storage system. A key feature behind Týr is its novel multi-versioning management designed to keep the metadata overhead as low as possible while still allowing fast queries or updates and preserving transaction semantics. Its share-nothing architecture ensures minimal contention and provides low latency for large numbers of concurrent requests. Týr is the first blob storage system to provide sequential consistency and high throughput, while enabling unforeseen transaction support. Experiments with a real-life application from the CERN LHC show Týr throughput outperforming state-of-the-art solutions by more than 100%.

Key-words: big data, middleware, storage, cloud, metadata, blobs, distributed systems, transactions, monitoring, consistency, grid5000

* Universidad Politécnica de Madrid, {pmatri,jmontes,mperez}@fi.upm.es
† IRISA / INSA Rennes, KerData, alexandru.costan@irisa.fr
‡ Inria Rennes, KerData, gabriel.antoniu@inria.fr

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Týr: Stockage Massif Transactionnel à Hautes-Performances

Résumé : À mesure que la puissance de calcul utilisée par des applications à grande échelle augmente, le volume de données qu'elles manipulent tend à augmenter également. Une grande partie de ces applications nécessite un système de stockage robuste et flexible permettant l'exécution de transactions de manière concurrente. Antérieurement, les bases de données furent la solution *de facto* pour la gestion des données transactionnelles, mais elles empêchent les applications de contrôler l'organisation du stockage des données ainsi que l'accés à ces données, tout en restant incapables de répondre aux contraintes posées par les données massives. Plus récemment, des systèmes de stockage clé-valeur ont été créés pour répondre à cette problématique. Cependant, ces solutions ne fournissent pas de support des transactions, ou seulement un support partiel, imposant aux utilisateurs de coordonner avec soin l'accès aux données afin d'éviter tout état de concurrence, écritures partielles, surécritures, ainsi que d'autres problèmes à l'origine d'un comportement erratique des applications. Nous soutenons qu'il existe un fossé entre les solutions de stockage actuelles et les besoins des utilisateurs, ce qui limite la conception des applications transactionnelles gérant des volumes massifs de données. Dans ce document, nous présentons Týr, un système de stockage de blobs distribué et transactionnel. Une des caractéristiques principales de Týr est sa gestion des versions novatrice conçue pour permettre un accès rapide tant en lecture qu'en écriture aux données tout en gardant une sémantique transactionnelle et en nécessitant une faible surcharge de métadonnées. Son architecture décentralisée garantit une contention minimale et permet une faible latence avec un nombre important de requêtes concurrentes. Týr est le permier système de stockage de blobs à fournir à la fois une consistence séquentielle et un débit élevé, tout en apportant le support des transactions. Les expériences réalisées avec une application réelle du CERN LHC montrent que le débit de Týr surpasse celui des solutions actuelles de plus de 100%.

Mots-clés : données massives, intergiciel, stockage, cloud, métadonnées, blobs, systèmes distribués, transactions, supervision, consistence, grid5000

1 Introduction

The recent development of the Internet of Things, the tremendous success of cloud-computing as a support for data-intensive scientific applications and the rapid growth of sensor and social networks have progressively emphasized the need for storage solutions capable of handling the extremely large volume of data generated by these systems. Twitter, for example, is processing and storing more than 500 million tweets a day, generated by more than 300 million users [1]. Considering the size (varying from a few Kilobytes to several Terabytes) and the high rate at which this data is generated, storing it efficiently becomes an important challenge.

Traditional databases simply cannot cope with such huge volumes arriving at fast rates with an inherently high complexity. This is partly due to the fact that this paradigm has originally not been designed with *horizontal scalability* in mind. To overcome such performance scalability limits, key-value stores such as Dynamo [2] or Cassandra [3] have been introduced. They are mainly targeted at storing immutable objects, as they typically only allow to replace the value of a key as a whole. To efficiently cope with mutable objects, a new generation of distributed file systems like HDFS [4] or Ceph [5] have emerged: they are based on the same principles and are designed to store large and mutable chunks of data.

More recently, we have witnessed the development of simpler data storage systems, targeting specific challenges, and optimized for a single use case. For instance, Twitter developed the FlockDB graph database [6] specifically for storing their social graphs.

In order to leverage the existing advances in data management, modern data-intensive applications demand more flexibility at the storage layer. This flexibility is essentially achieved by making the least possible assumptions on the data size, structure and access patterns. Binary Large Objects, or blobs, are a perfect illustration of this principle: they allow storing unstructured data accessed through low-level binary methods. Their unstructured nature makes them good candidates for efficiently storing large numbers of related events by grouping them in a single storage container and avoiding the cost of storing metadata for each individual object.

However, while existing blob-based storage systems like BlobSeer [7] are able to gracefully scale up and achieve high access throughput, they are typically getting this performance at the expense of weaker consistency guarantees. Yet, *sequential consistency*[1] is a crucial requirement for many operations related to data-intensive applications (e.g. checkpoint/restart, data indexing, snapshotting etc.). Traditional databases usually meet this requirement by means of *transactions*. Their semantics have been transposed to more general distributed storage systems through heavyweight synchronization protocols that significantly affect the performance of the storage system. Providing such a strong consistency while meeting strict performance requirements has proved to be a challenging task given the high level of coordination needed for transaction processing. In order to efficiently provide full transaction semantics, such as *success or failure as a complete unit*, this feature needs to be integrated in the storage layer rather than delegating it to a an upper middleware layer (as in ZooKeeper [9] for instance).

In this paper, we introduce Týr, a *high-performance distributed blob storage system* with *built-in lightweight transactions*. Týr aims to serve as a flexible storage backend for a wide variety of data-intensive applications with strong performance requirements. It allows for fast, *single-hop queries* (i.e. direct queries to the nodes holding the data) under heavily concurrent access, while a novel version management scheme provides low-overhead data storage. Designed with flexibility in mind, Týr is able to cope with arbitrarily large datasets (ranging from small data in the order of Bytes and Kilobytes to big data in the order of Terabytes) and to gracefully

[1] Guarantee that the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and that the operations of each individual processor appear in this sequence in the order specified by its program [8].

scale-out on a cluster of commodity machines. The contributions set forth in this paper can be summarized as follows:

- **A set of innovative version management techniques** designed to keep the metadata overhead as low as possible while still allowing fast queries or updates and preserving transaction semantics.
- **A novel blob storage architecture** based on the above versioning techniques, providing at the same time *lightweight transaction semantics* and *high performance*. This architecture has been implemented in a software prototype and validated using a real life scenario, as a storage backend for the MonALISA [10] monitoring system of the CERN LHC ALICE experiment [11].
- **An experimental study** showing that Týr's throughput outperforms existing solutions such as HDFS [4] and BlobSeer [7] by up to 104%, while preserving the sequential consistency guarantees and using a flexible data model.

This paper is structured as follows. Section 2 illustrates in a more detailed way the challenges we address, thanks to a real-world scenario. Section 3 details the proposed architecture, illustrated through the Týr prototype described in Section 4. Section 5 presents the results of our experimental evaluation, followed by a discussion of some other related work in Section 6. Finally, Section 7 concludes this paper and outlines future work on Týr.

2 Motivating scenario

To better understand the needs in terms of storage for real-world large-scale scientific applications, we considered the case of ALICE (A Large Ion Collider Experiment) [11], one of the four LHC (Large Hadron Collider) experiments realized at CERN (European Organization for Nuclear Research) [12]. Its scale, volume, collection rate and geographical distribution of data require appropriate tools for efficient storage. The ALICE collaboration, consisting of more than 1,000 members from 29 countries and 86 institutes, is indeed strongly dependent on a distributed computing environment to perform its physics program. The experiment collects data at a rate of up to 4 Petabytes and produces more than 10^9 data files per year. Tens of thousands of CPUs are required to process and analyze them. The CPU and storage capacities are distributed over more than 80 computing centers worldwide.

We focus on the management of the monitoring information collected in real-time about all ALICE resources. More than 350 MonALISA services are running at sites around the world, collecting information about ALICE computing facilities, local and wide area network traffic, and the state and progress of the many thousands of concurrently running jobs. This yields more than 1.1 million parameters published in MonALISA, each with an update frequency of one minute. Using ALICE-specific filters, the raw parameters are aggregated to produce about 35,000 system-overview parameters in real time. This monitoring data needs to be efficiently stored for further processing: it is indexed using timestamps and requires *transaction support*, as monitoring data is *concurrently updated and read* by multiple processes.

Starting from this use-case, we summarize the key requirements of a storage system supporting such large-scale applications:

- **Flexible data model.** The system should support efficient storage and access to both small and large unstructured objects in the form of blobs.
- **High-throughput under heavy concurrency.** The high rate at which concurrent events are generated calls for a storage layer able to support parallel data processing to a high degree, over a large number of nodes. With potentially thousands of clients simultaneously accessing data, the storage must sustain a high throughput in spite of a high level of concurrency, while at the same time providing efficient fine-grain access to data.
- **Built-in transaction support.** Applications heavily relying on data indexing as well as live computation of aggregates (e.g. derivate metrics in monitoring systems like MonALISA) require a transactional storage system able to synchronize read-update operations and to guarantee the consistency of the data.
- **Performance stability.** The storage system permanently collects data, which has to be kept over the entire lifespan of the applications. Stable and predictable storage performance are crucial for the efficiency of the subsequent data processing.
- **Horizontal scalability.** In order to cope with always increasing datasets, the storage system must be able to scale out and expand its capacity. This should be possible at any time without interrupting the data collection.

These are precisely the requirements addressed by the Týr approach, described in the following section.

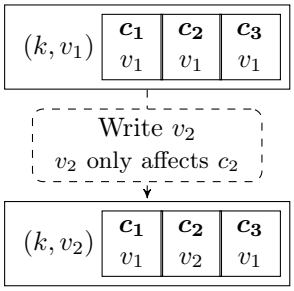


Figure 1: Týr versioning model. When a version v_2 of the blob is written, which only affects chunk c_2, only the version of both the blob and c_2 is changed. The version id for both c_1 and c_3 remains unchanged.

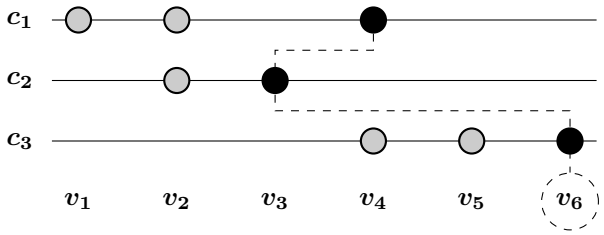


Figure 2: Version management example. The version v_1 of this blob only affected the chunk c_1, v_2 affected both c_1 and c_2. In this example, v_6 is composed of the chunk versions (v_4, v_3, v_6). This versioning information is stored on the metadata nodes for this blob.

3 Our proposal: the Týr approach

We are introducing Týr, a distributed storage system for raw, unstructured binary objects (as opposed to structured data). In this section, we detail the design and architecture principles making Týr able to address the previous requirements.

3.1 Data striping and replication

The first requirement for Týr is to be able to handle a large number of concurrent reads and writes. To achieve this goal, *data striping* is used to balance read and writes over a large number of nodes in parallel. Blobs in Týr are splitted in multiple chunks of a size defined for the whole system. With a chunk size s, the first chunk c_1 of a blob will contain the bytes in the range $[0, s)$, the second chunk c_2, possibly stored on another node, will contain the bytes in the range $[s, 2s)$, and the chunk c_n will contain the bytes in the range $[(n-1) * s, n * s)$. In order to distribute the reads even more, each chunk is replicated to additional nodes in the cluster.

3.2 Distributed Hash Table based data distribution

One of the key goals of Týr is the *horizontal scalability*. This requires the ability to dynamically partition the data over the set of nodes (i.e. storage hosts) in the cluster.

Týr distributes both metadata and data across the cluster using consistent hashing [13], based on a *distributed hash table*, or DHT. Given a hash function $h(x)$, the output range $[h_{min}, h_{max}]$ of

the function is treated as a circular space (h_{min} sticking around to h_{max}). Every node is assigned a different random value within this range, which represents its position on the ring. Each data chunk n of a blob with key k is stored on a node determined by hashing the concatenation of k and n using $h(k : n)$, giving a unique position h_k on the ring. The nodes on which the additional replicas of the element will be stored are obtained by continuing walking the ring passed the coordinator node until we find the appropriate number of nodes eligible to hold a replica of this data given the replication settings of the system.

The main advantage of consistent hashing is that departure or arrival of a node only affects its immediate neighbors. Other nodes remain unaffected. The basic algorithm is oblivious to the heterogeneity in the performance of nodes. To address this issue, each node gets assigned different positions on the ring, as implemented in Dynamo [2].

A Týr client may query any node in the cluster for the current state of the ring to be able to address its queries to the appropriate node directly.

3.3 Version management

In order to achieve high write performance under concurrent workloads, Týr uses *multi-version concurrency control*. This ensures that the current version of a blob can be read while a new one is being created without locking. Version management is done implicitly in Týr. Version identifiers are internal to Týr and are not exposed to the calling application.

When a client writes data to some chunk of a given blob, a random version id is generated by the server. A new version of the blob chunks affected by the operation is generated, and tagged with that version number. The other chunks remain unchanged, as illustrated by Figure 1.

During a write operation, the nodes holding unaffected data chunks will not receive information regarding the new version. Consequently, the latest version of a blob is composed of a set of possibly different versions of its chunks, as depicted in Figure 2. Thus, it is necessary to be able to construct the right set of chunk versions when accessing a specific version of a blob. For that matter, the information regarding successive versions of each chunk is stored on the same nodes holding replicas of the first data chunk of the blob. These nodes are called *metadata nodes* for the current blob. The full write protocol is explained in Section 3.6.

3.4 Basic API

We propose an asynchronous set of primitives for Týr. Rather than blocking the calling application while the action is performed, control is returned directly once the request has been sent. A callback function passed as argument will be executed when the action has been performed with the results of the operation.

```
CREATE(key, callback(status))
```

This function creates an empty blob of size 0, identified by a *key* chosen by the client. The operation will fail if a blob with the same key already exists in the system.

```
WRITE(key, offset, length, buffer, callback(status))
```

This function writes the first *length* bytes of *buffer* to the blob identified by *key* at *offset*.

```
READ(key, offset, length, callback(status, buffer))
```

This function reads *length* bytes of the blob identified by *key* at *offset*.

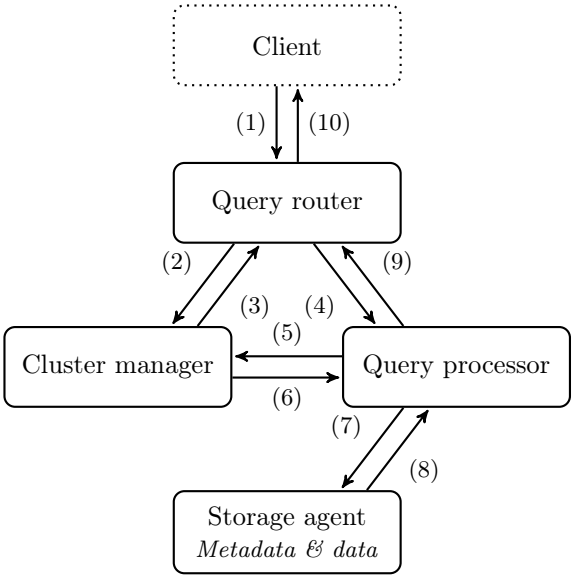


Figure 3: Týr server high-level architecture.

```
TRUNCATE(key, length, callback(status))
```

This function truncates the size of the blob identified by *key* to *length* bytes. If the blob was previously larger than *length*, the extra data is lost. If the blob was previously shorter, it is extended, and the extended part reads as null bytes.

If transaction semantics are needed, the following functions may be used in complement to the primitive functions:

```
BEGIN()
```

This synchronous function creates a local transaction context, which does not involve any communication with the cluster. Every subsequent read or write operation will use that context.

```
ROLLBACK()
```

This synchronous function discards the currently open local transaction context without applying the changes.

```
COMMIT(callback(status))
```

This function closes the open local transaction context applying the changes. The operation will fail if any conflicting operation has been executed concurrently. In case of any failure, the entire operation will rollback and fail.

3.5 System architecture

Týr has been developed following a modular and loosely coupled architecture. Each node runs a set of different modules, executed on different threads. Figure 3 shows the interaction between the modules of the system, which are:

- **The Cluster Manager** maintains the ring state across the cluster using a weakly-consistent gossip protocol [14] to propagate information around the cluster (i.e. ring position allocations) and a ϕ accrual failure detector [15] to detect and confirm failures in the cluster.
- **The Query Processor** is the heart of the system. For write queries, it coordinates the requests, and is responsible of handling the transaction protocol using cluster information from the cluster manager *(5, 6)*. It acts as the interface to the storage agent *(7, 8)*.
- **The Query Router** is the main communication interface between a server and both the rest of cluster and the clients. Its responsibility regarding incoming client requests *(1)* is to receive, parse, validate and, if necessary, to forward them to the appropriate server according to cluster state information *(2, 3)*. It then forwards the requests to the query processor *(4, 9)* and finally responds to the client *(10)*.
- **The Storage Agent** is responsible for the persistent storage and retrieval of both data and metadata.

3.6 Write protocol

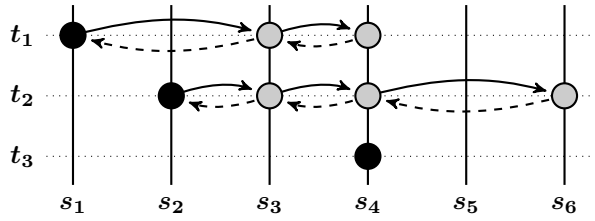
Because both the data chunks and the metadata are stored on different nodes in the cluster and are getting replicated, we need to ensure that the ordering of two concurrent writes is consistent among affected nodes of the cluster. To this purpose, Týr uses the Warp transaction protocol [16], introduced for the HyperDex [17] key-value store. The base protocol has been extended to take into account the presence of metadata nodes.

Every write request, included or not in a transaction on the client, is processed as a transaction at the server. In order to execute a transaction, the client constructs a chain of servers which will be affected by it. These nodes are all the ones storing the written data chunks, one node holding the data for each chunk read during the transaction (if any), plus all metadata nodes for the affected blobs. This set of servers is sorted in a predictable order, such as a bitwise ordering on the IP / Port pair. This order ensures that conflicting transactions pass through their shared set of servers in the exact same order. If the node to which the request has been sent is not the first one in the chain, it will pass the request to that node. The first node in this chain is called *coordinator node* for that request.

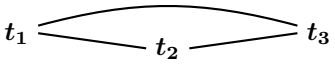
The client then forwards the request to the coordinator node. This node will validate the chain and ensure that it is up-to-date according to the latest ring status. If not, that node will construct a new chain, and forward the request to the coordinator node of the new chain, in case it has changed.

We use a linear transactions commit protocol to guarantee that all transactions are either successful and serializable, or abort with no effect. This protocol consists of one forward pass to optimistically validate the values read by the client and ensure that they remained unchanged by concurrent transactions, followed by a backward pass to propagate the result of the transaction – either success or failure – and actually commit the changes to memory. Dependency information is embedded by the nodes in the chain during both forward and backward passes to enforce a serializable order across all transactions. A background garbage collection process limits this number of dependencies by removing those that have completed both passes.

The forward pass validates transactions by ensuring that new transactions do not conflict nor invalidate previously validated transactions, for which the backward pass is not complete. Every node in the commit chain ensures that the transactions do not read values written by, or write values read by previously validated transactions. Nodes also check each value against the latest one stored in their local memory to verify that the data was not changed by a previously



(a) Propagation graph



(b) Dependency graph

Figure 4: Týr write protocol. In that example, the directionality of the edge (t_1, t_2) will be decided by s_4, last common server in the transaction chains, during the backwards pass. Similarly, the directionality of (t_2, t_3) will be decided by s_4.

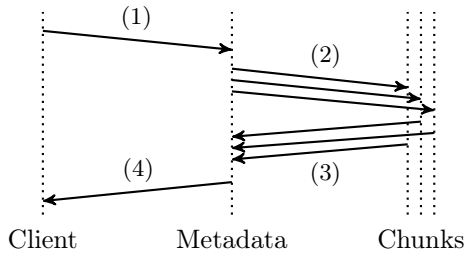


Figure 5: Týr general read protocol.

committed transaction. The validation step fails if transactions fail either of these tests. A transaction is aborted by sending an abort message backwards through the chain members that previously validated the transaction. These members remove the transaction from their local state, thus enabling other transactions to validate instead.

Servers validate each transaction exactly once, during the forward pass through the chain. As soon as the forward pass is completed, the transaction may commit on all servers. The last server of the chain may commit the transaction immediately after validating it, and sends the commit backwards to the chain.

Enforcing a serializable order across all transactions requires that the transaction commit order does not create any dependency cycles. To this end, a local dependency graph across transactions is maintained at each node, with the vertices being transactions and each directed edge specifying a conflicting pair of transactions. A conflicting pair is a pair of transactions where one transaction writes at least one data chunk read or written by the other.

Whenever a transaction validates or commits after another one at a node, this information is added to the transaction message sent through the chain: the second transaction will be recorded as a dependency of the first. This determines the directionality of the edges in the dependency graph. A transaction is only persisted in memory after all of its dependencies have committed, and is delayed at the node until this condition is met.

Figure 4 illustrates this protocol with an example set of conflicting transactions chains, and the associated dependency graph. This example shows three transaction chains executing. Figure 4a shows the individual chains with the server on which they execute. The black dot represents the coordinating server for each transaction, the plain lines the forward pass, and the dashed lines the backwards pass. Because they overlap at some servers, they form conflicting pairs as shown on the dependency graph in Figure 4b. The directionality of the edges will be decided by the protocol, as the chain is executed on the servers.

3.7 Read protocol

Reading a blob in Týr is a simple process: every read, included or not in a transaction on the client-side, is processed exactly the same way.

3.7.1 General case

The version management introduced previously causes each node holding data for a specific blob to have a different and incomplete knowledge of its successive versions. This implies that a metadata node has to be queried for any read.

Figure 5 illustrates the different steps necessary to read data from Týr. The client addresses the read query to one of metadata nodes for the blob from which to read *(1)*. Based on the version information it holds, the metadata node determines the latest set of chunk versions for every chunk read by this request. It then forwards the read request to a randomly chosen replica for each chunk, along with the appropriate version for that chunk *(2)*. As soon as all the data has been received from the chunk replicas *(3)*, it forwards it to the client *(4)*.

3.7.2 Direct read

The transaction protocol allows each node to know the latest version of every data chunk it stores. As an optimization to the protocol introduced in the previous section, a read request outside of any client-side transaction that can be answered by reading exactly one data chunk can bypass the metadata node. The client can then address the request directly to one of the nodes holding the data. The queried node returns to the client the latest confirmed successfully written version.

3.8 Ring coordination

The ring coordinator is the process that assigns to each node of the cluster the token ranges on the ring it will be responsible for. This process is unique in the whole cluster. It is automatically started on the first server joining the cluster. The targeted *horizontal scalability* requires Týr to be able to seamlessly integrate new nodes to an already running cluster. Whenever a new node joins the cluster, the ring coordinator will assign it a share of the token range taken from the existing servers. These token ranges are marked as being moved on all servers previously responsible for them.

Data is then moved in the background from the old nodes to the new ones. Every running transaction writing data on these ranges will include the new nodes as well as the old ones. Readers will continue reading on the other nodes during the moving process.

Once all data has been copied, the information is pushed to all nodes in the cluster. After each node acknowledged the new token ring state, the data is removed from the old nodes. Should this process fail in the cluster, it is immediately restarted on another node chosen randomly.

3.9 Versioning policy

The removal of older versions is performed progressively. During general-case read requests, the metadata node queried for the current version state of a blob keeps track of the blob version it responds with. When a write request for a blob arrives at a node holding the associated metadata, both the oldest version of the blob in use and the associated chunk versions are piggybacked to the validation message during the forward pass of the write protocol. Each subsequent node in the chain holding metadata for the same blob replaces these versions by the local oldest versions if the blob version is older than the one bundled in the commit message. During the backwards pass, each node in the chain is then free to remove any data for versions older than the ones indicated in the commit message.

4 Týr prototype implementation

All the features of Týr relevant for this paper have been fully implemented. This implementation includes the Týr server, the stratified version management library, the C client library, as well as partial C++ and Java bindings. The server itself is approximately 10,000 lines of GNU C code. This section describes key aspects of the implementation.

4.1 General implementation details

Týr is optimized for low-latency and high-throughput query processing. To meet these requirements, it makes heavy use of the multiprocessing capabilities of current computers, of lock-free data structures whenever possible, and of event-driven programming.

Týr is internally structured around multiple, lightweight and specialized event-driven loops, backed by the LibUV library [18]. When a request is received, it is forwarded for processing to one of the relevant event loops for further asynchronous processing. No request queuing is done in order to avoid communication delays, and thus reduce the overall latency of the server. On-disk data and metadata storage is built using Google's LevelDB key-value store [19], a state-of-the-art log-structured merge tree [20] based library optimized for performance on hard disks.

Special care has been taken on the design of the data structures used. Whenever possible, Týr makes use of lock-free, optimistic data structures based on atomic operations to avoid lock-contention in performance-critical parts of the server. In order to further improve the global performance, these data structures are designed to be cache-oblivious, taking advantage of the CPU cache as much as possible. The intra-cluster and client-server request / response messages are serialized using the Cap'n'proto [21] library.

4.2 Implementation details on version management

The object identifiers are generated using MongoDB's ObjectId format [22]. ObjectIds are 12-byte non-sequential and globally-unique identifiers generated from the current system date, process information and a local counter.

The version management data structure itself is based on multiple linked lists. One structure is allocated for each blob. One list is used to keep track of the successive versions, and one additional list is used for each of the chunks composing a blob. Each time a new version is created, a new entry is added in the version tracking list, plus one entry in the lists covering every affected chunk. A copy-on-write guard structure always points to the latest version of the blob and all of its chunks. Both reading the chunks versions for the latest version of the blob and adding a new version to the structure are $O(1)$ operations. Additionally, because of the atomic nature of this structure, writers do not block readers.

5 Evaluation

We evaluated our design in five steps. We first studied the transactional write performance of a Týr cluster with a heavily-concurrent usage pattern. Second, we tested the raw read performance of the system. We then gauged the reader / writer isolation in Týr. Fourth, we measured the performance stability of Týr over a long period. Last, we proved the *horizontal scalability* of Týr.

Experimental setup. We deployed Týr on the Grid'5000 [23] experimental grid testbed, distributed over 11 different sites in both France and Luxembourg. For these experiments, the *paravance* cluster of Rennes was used. Each node is outfitted with 2 x 2.4 Ghz octo-core Intel Xeon E5-2630v3 processors, 128 GB of RAM and 10 Gigabit Ethernet connectivity (MTU = 1500 B).

Evaluated systems. To the best of our knowledge, no distributed storage systems with a comparable low-level data model and built-in transactions are available today. As such, throughout these experiments, we compared Týr with HDFS [4], a *de facto* standard distributed file system available today on most cloud data analytics platforms as part of the Hadoop [24] stack. HDFS stores the metadata information on a central metadata node. We also compared Týr with BlobSeer [7], an open-source, in-memory distributed storage system which shares the same data model and a similar API. BlobSeer has been designed to support a high-throughput for highly-concurrent accesses to shared distributed blobs, and was shown to outperform HDFS, therefore being a relevant basis for comparison. BlobSeer distributes the metadata over the cluster by using a distributed tree. For these experiments, we used BlobSeer 1.2.1 and Hadoop 2.7.1.

Metrics. The main metric we were interested in is the *aggregate throughput*. The throughput is the number of operations processed (either reads or writes, depending on the test) per unit of time. In the case of both Týr and BlobSeer, the API is asynchronous. As such, for these two systems, we are considering the callback invocation as the response. For each of these tests we ran the experiments 50 times under the same conditions, and averaged the results obtained. Overall, our measurements showed little variance and were stable enough to be conclusive.

Dataset and workload. In order to run these experiments, we used a dump of real data obtained from the MonALISA system [10]. This data set is composed of ~ 4.5 million individual measurement events, each one being associated to a specific monitored site. We used multiple clients to replay these events, each holding a different portion of the data. We stored the events on one blob (or file for HDFS), depending on the origin of the event, and maintained an append-only data index on two other blobs (or files). The read tests were performed by querying ranges of data, simulating a realistic usage of the MonALISA interface. In order to further increase read concurrency, the queried data was randomly selected using a power-law distribution.

5.1 Transactional write performance

High transactional write performance is the key requirement that guided the design of Týr. To benchmark the different systems in that context, we measured the transactional write performance of Týr, BlobSeer and HDFS with the MonALISA workload. Transactions are required to synchronize the storage of the events and their indexation in the context of a concurrent setup. Because of the lack of native transaction support in both BlobSeer and HDFS, we used ZooKeeper [9], an industry-standard distributed synchronization service. Being part of the Hadoop stack, as HDFS, it came as a natural choice. To avoid any bias in our comparison, we used ZooKeeper on top of both BlobSeer and HDFS. This system has been used as a middleware to synchronize the writes operations in the cluster based on distributed locks. Because

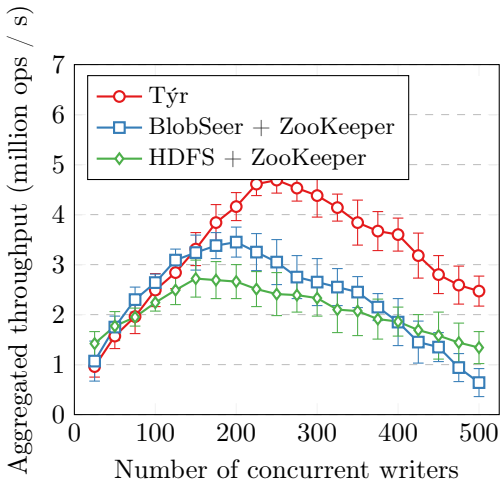


Figure 6: Write performance of Týr, BlobSeer and HDFS on a 16-node cluster, varying the number of concurrent clients, with 95% confidence interval.

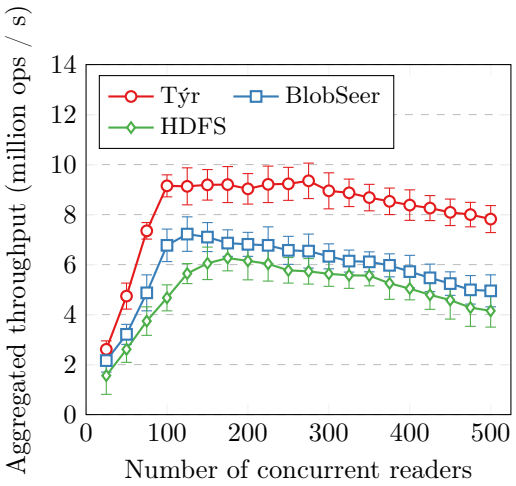


Figure 7: Read performance of Týr, BlobSeer and HDFS on a 16-node cluster, varying the number of concurrent clients, with 95% confidence interval.

Týr transactions conflicts are detected at the chunk level, one lock is used for each data chunk in the system.

The results, depicted on Figure 6, show that Týr outperforms HDFS and BlobSeer under high concurrency, and topped at 4.58 million operations per second on our experimental setup. The clear decrease in performance on higher concurrency in the case of BlobSeer and HDFS is due to the higher ZooKeeper lock contention. The built-in transaction support of Týr allows it to scale better. However, under high concurrency, conflicts between transactions are causing processing delays that limit the performance of the cluster. It is worth to note that the additional network queries need to maintain the metadata tree in the case of BlobSeer, having a high impact on the network performance, which explains the faster decrease in performance compared to HDFS.

With lower concurrency, however, we can see that the transaction protocol incurs a slight processing overhead, making all three systems perform with comparable performance when the lock contention is low.

5.2 Read performance

Týr introduces important optimizations for read performance, most notably single-hop and lock-free reads for a recent version of a given chunk. In order to demonstrate the advantages of this feature, we evaluated the read performance of Týr and compared it with the results obtained with BlobSeer and HDFS on the same setup. We preloaded in each of these systems the whole MonALISA dataset, for a total of around 100 Gigabytes of uncompressed data. We then performed random reads of 100 Kilobytes size each from both the raw data and the aggregates, following a power-law distribution to increase read concurrency on the cluster.

We plotted the results in Figure 7. The lightweight read protocol of Týr allows its aggregated performance to quickly exhaust the network bandwidth at 9.19 Million operations per second, and allows it to outperform both BlobSeer and HDFS. The whole bandwidth, except for gossip cluster state messages, was used for client-server communication. In contrast, both BlobSeer

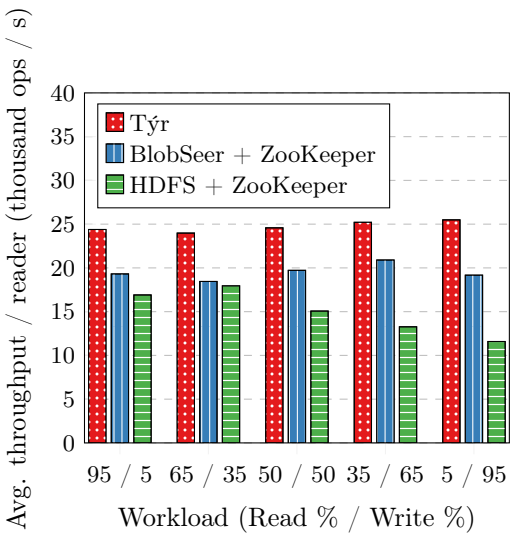


Figure 8: Throughput for different data sizes. Each bar represents the average throughput of a steady 16-node cluster with 250 concurrent clients averaged over a one-minute window, for workloads with different settings of read to write ratio.

and HDFS require multiple hops to fetch the data, either to the metadata node for HDFS, or in the distributed metadata tree for BlobSeer. This incurs an additional networking cost that limits the total performance of the cluster. Under higher concurrency, we observe a slow drop in throughput for all the compared systems, due to the involved CPU in the cluster getting overloaded.

5.3 Reader / writer isolation

We used *multi-version concurrency control* to ensure that writers never block readers in the cluster. This is a key feature for any write-oriented usage pattern. Týr has been designed accordingly.

We performed simultaneously reads and writes in the cluster, using the same setup and methodology as with the two previous experiments. To that end, we preloaded half the data in the cluster, and measured read performance while writing the rest of the data. We ran the experiments using 250 concurrent clients. On that configuration, all three systems proved to perform above 85% of their peak performance for both reads and writes, thus giving comparable results and a fair comparison between the systems. Among these clients, we varied the proportions of readers and writers in order to measure the performance impact of different usage scenarios. For each of these experiments, we were interested in the average throughput per reader.

The results, depicted in Figure 8, confirm that Týr outperforms its competitors for all five usage patterns tested, as showed in the previous subsection. This is the result of the single-hop read feature of Týr. It also demonstrates the added value of *multi-version concurrency control*, on which both Týr and BlobSeer are based. For these two systems, we observe an average read performance per writer which is independent of the number of concurrent writers. In the case of HDFS, we can observe the significant impact of writers on the read performance of the system, due to the internal locking required to ensure concurrency control.

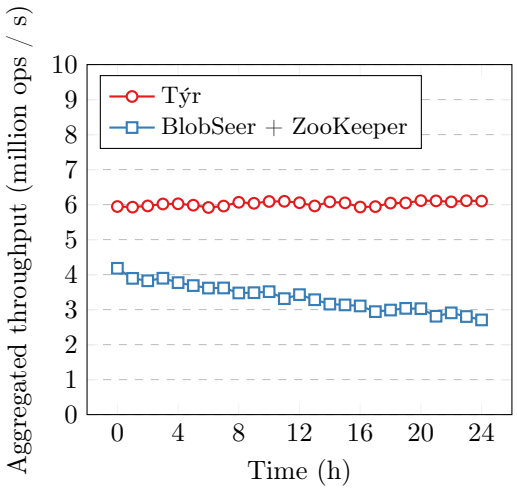


Figure 9: Comparison of aggregated read and write performance stability over a 24-hour period in a 16-node cluster with a 65% read / 35% write workload.

5.4 Performance stability

Týr data structures have been carefully designed so that successive writes to the cluster impact access performance as little as possible. We validated this behavior over a long period of time using a 16-node setup on the *paravance* cluster, and 250 concurrent clients. We have used a 35% write / 65% read workload, which is the most common workload in MonALISA.

Due to storage limitations on the nodes, it was not possible to perform writes at maximum speed for a long enough period of time. Therefore, we chose to write the same MonALISA data repeatedly, overwriting the old data on each pass. The lack of support for random writes in HDFS [25] prevented us from including it in this evaluation. As our previous experiments showed that BlobSeer outperforms HDFS in the same setup, this is not going to affect the conclusions derived from this scenario.

We depict in Figure 9 the average throughput per writer over an extended period of time for both reads and writes. The results confirm that Týr performance is stable over time. On the other hand, BlobSeer shows a clear performance degradation over time, which we attribute to its less efficient metadata management scheme: for each blob, metadata is organized as a tree that is mapped to a distributed hash table hosted by a set of metadata nodes. Accessing the metadata associated with a given blob chunk requires the traversal of this tree; as the height of this tree increases, the number of requests necessary to locate the relevant chunk metadata also increases. This results in a more important number of round-trips between the client and the server, and consequently in a degraded performance over time. During this experiment, we also measured the isolated performance of ZooKeeper. The collected figures show a constant performance over time. Thus, we ruled out its influence in the progressive performance decrease of BlobSeer.

5.5 Horizontal scalability

Finally, we tested the performance of Týr with a cluster of increasing size. This results in an increased throughput as is distributes the load over a larger number of nodes. We used the same setup as for the previous experiment, varying the number of nodes and the number of clients,

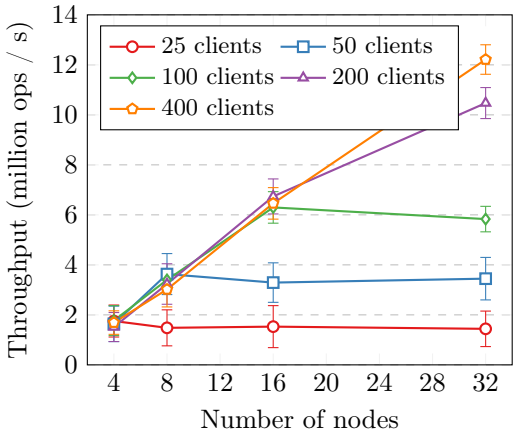


Figure 10: Týr *horizontal scalability*. Each point shows the average throughput of the cluster over a one-minute window with a 65% read / 35% write workload, and 95% confidence intervals.

and plotting the achieved aggregated throughput among all the clients over a one-minute time window. We have used the same 35% write / 65% read workload as in the previous experiment. Figure 10 shows the impact of the number of nodes in the cluster on system performance. We see that the maximum average throughput of the system scales linearly as new servers are added to the cluster.

6 Related work

While extensive research efforts have been dedicated to optimize the storage of Big Data (with sizes in the order of Giga- and Terabytes), there has been relatively less progress on identifying, analyzing and understanding the challenges of ensuring increased flexibility and expressivity at the storage level (e.g. enabling transactions, general data models, support for Small Data etc.). A new generation of storage and file systems as well as optimisations brought to existing ones, try to address these new challenges. However, as described below, they focus on very specific issues, in most cases trading performance for expressivity.

Disk Storage. Specialized file systems specifically target the needs of data-intensive applications. Inspired by Google's File System [26], HDFS, the standard storage layer used by Hadoop MapReduce [24], has become the reference for distributed disk-based storage of large files. Such systems essentially keep the data immutable, while allowing concurrent appends. While this enables exploiting data workflow parallelism to a certain degree, it greatly suffers from using locking to handle concurrency and from the centralized metadata management. BlobSeer [7] is a highly scalable distributed storage system from which Týr took great inspiration and with which it shares the same data model. BlobSeer introduces several optimizations among which metadata decentralization and versioning-based concurrency control that enable writes at arbitrary offsets, effectively pushing the limits of exploiting parallelism at data-level even further. Týr goes even further by eliminating the centralized version manager and distributing that information over the whole cluster. This feature allows Týr to scale linearly with the number of nodes in the cluster. Additionally, neither HDFS nor BlobSeer do support transactions or offer single-hop reads.

Key-Value Stores. Since disk-oriented approaches to online storage are unable to scale gracefully to meet the needs of data-intensive applications, and improvements in their capacity have far out- stripped improvements in access latency and bandwidth, recent solution are shifting the focus to random access memory, with disk relegated to a backup/archival role. Most open source key-value stores have roots in work on Distributed Data Structures [27] and distributed hash tables [13]. Similarly to Týr, they are inspired by the ring-based architecture of Dynamo [2], a structured overlay with at most one-hop request routing. Write operations in Dynamo require a read to be performed for managing the vector clock scheme used for conflict resolution, which proves to be a very limiting factor for high write throughput. BigTable [28] provides both structure and data distribution but relies on a distributed file system for its durability. Cassandra [3] takes inspiration from both BigTable (for the API) and Dynamo (for data distribution) to offer scalability and availability properties that traditional database systems simply cannot provide. Yet these properties come at a substantial cost: the consistency guarantees are often quite weak, it only supports single-hop reads and doesn't allow for random writes or even appends to values. The closest system to ours is Hyperdex [17], a key-value store in which objects with multiple attributes are mapped into a multidimensional hyperspace, instead of a ring. This mapping allows indexing and leads to efficient implementations not only for retrieval by primary key, but also for range queries. Similarly to Týr, a chaining protocol enables the system to achieve sequential consistency, and maintain availability. Hyperdex uses Warp [16] as an additional layer for providing ACID-like transactions on top of the store with minimal performance degradation. However, compared to Týr, HyperDex is a higher-level system that offers a lower control over the data layout, and does not allow to store mutable objects.

Metadata Optimisations. Distributing metadata across several servers mitigates the bottleneck of centralised management for systems involving access to many small files, when the latency for metadata access can become dominant on the overall data access time. Currently, there are two major methods used to distribute the namespace and workload among metadata servers in existing distributed file systems: partitioning and hashing.

Namespace Partitioning provides a natural way to partition the namespace among multiple servers according to directory subtrees. In dynamic partitioning, metadata is automatically partitioned and distributed to servers according to the specific load-balancing policy of the filesystem. If some metadata severs become overloaded, some of its metadata subtrees could be migrated to other servers with light load. While providing better scalability this might cause slow metadata lookup. Several state-of-the-art file systems rely on this partitioning technique. Ceph [5] dynamically scatters sets of directories based on server load. PVFS [29] uses a fine-grained namespace distribution by scattering different directories, even those in the same sub-tree, on different metadata servers. PanFS [30] is more coarse-grained: it assigns a subtree to each metadata server. Giraffa [31] leverages HBase [32], a distributed key value store, to achieve load balancing on directory collections, suffering from the aforementioned hot entries issue.

Hashing eliminates the issue of unbalanced workload among nodes by assigning metadata based on a hash of the file identifier, file name or any related value. Although with this approach metadata can be distributed uniformly, the directory locality feature is lost, and if the path is changed, some metadata has to migrate. Also, a directory hierarchy must be maintained and traversed in order to support standard file naming semantics, tampering some of the apparent benefits. Lustre [33] makes a hash on the tail of the filename and the identifier of the parent directory to map the metadata to a server. CalvinFS [34] uses hash-partitioned key-value metadata across geo-distributed datacenters to handle small files, with operations on file metadata transformed into distributed transactions. However, in contrast to Týr, this system only allows operations on single files. Multi-file operations require transactions. The underlying metadata database can handle such operations at high throughput, but the latency of such operations tends to be higher than in traditional distributed file systems.

Overall, most of the previous work typically focuses on some specific issues in isolation: either optimising metadata access or collective operations, or, if the work targets in-memory storage the focus falls on specific low-level issues that are not necessarily well correlated with the higher level design (e.g. trading consistency for performance). This is precisely this gap we aim to address in this work: Týr sets a new bar for future blob storage systems by combining sequential consistency properties with transaction support a rich API and high performance.

7 Conclusion and future work

In this paper, we have introduced Týr, a novel high-performance, transactional blob storage system. Both data and metadata are distributed throughout the cluster using a distributed hash table, without any centralized process for reads and writes. Multi-version concurrency control provides high-performance under heavily concurrent usage patterns, while a lightweight transaction process is used to ensure serializability of the writes and consistency across the cluster, being also exposed to the clients. Experiments on real-world data show that Týr outperforms state-of-the-art systems by more than 100%, while additionally enabling transaction support, preserving the sequential consistency guarantees and using a flexible data model.

Since the storage of large volumes of data typically spans over multiple data centers, we plan to extend Týr to support this kind of deployment. In this context, we will explore the possibility to further enhance read performance by introducing a dynamic, location- and usage-aware protocol. As an additional layer, we are also investigating the development of a distributed transactional file system using Týr as its storage backend.

Acknowledgment

This work is part of the BigStorage project, funded by the European Union under the Marie Skłodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). The authors would like to thank Costin Grigoras (CERN) for his valuable support and insights on the MonALISA monitoring traces of the ALICE experiment.

References

[1] "Twitter," 2015. [Online]. Available: https://about.twitter.com/company

[2] 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," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[4] J. Shafer, S. Rixner, and A. Cox, "The Hadoop distributed filesystem: Balancing portability and performance," in *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, March 2010, pp. 122–133.

[5] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI*, 2006, pp. 307–320.

[6] "Introducing FlockDB," 2015. [Online]. Available: https://blog.twitter.com/2010/introducing-flockdb

[7] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 169 – 184, 2011.

[8] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[9] "Apache ZooKeeper," 2015. [Online]. Available: https://zookeeper.apache.org/

[10] I. Legrand, H. Newman, R. Voicu, C. Cirstoiu, C. Grigoras, C. Dobre, A. Muraru, A. Costan, M. Dediu, and C. Stratan, "MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems," *Computer Physics Communications*, vol. 180, no. 12, pp. 2472 – 2498, 2009.

[11] The ALICE Collaboration, K. Aamodt, A. A. Quintana, R. Achenbach, S. Acounis, D. Adamová, C. Adler, M. Aggarwal, F. Agnese, G. A. Rinella, Z. Ahammed, A. Ahmad, N. Ahmad, S. Ahmad, A. Akindinov, P. Akishin, D. Aleksandrov, B. Alessandro, R. Alfaro, G. Alfarone, A. Alici, J. Alme, T. Alt, S. Altinpinar, W. Amend, C. Andrei, Y. Andres, A. Andronic, G. Anelli, M. Anfreville, V. Angelov, A. Anzo, C. Anson, T. Anticić, V. Antonenko, D. Antonczyk, F. Antinori, and S. A. et al., "The ALICE experiment at the CERN LHC," *Journal of Instrumentation*, vol. 3, no. 08, p. S08002, 2008.

[12] "CERN," 2015. [Online]. Available: http://home.cern/

[13] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663.

[14] A. Das, I. Gupta, and A. Motivala, "Swim: scalable weakly-consistent infection-style process group membership protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, 2002, pp. 303–312.

[15] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The ϕ accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, Oct 2004, pp. 66–78.

[16] R. Escriva, B. Wong, and E. G. Sirer, "Warp: Lightweight multi-key transactions for key-value stores," Cornell University, Tech. Rep., 2013.

[17] ——, "Hyperdex: A distributed, searchable key-value store," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12. New York, NY, USA: ACM, 2012, pp. 25–36.

[18] "LibUV," 2015. [Online]. Available: https://github.com/libuv/libuv

[19] "LevelDB: A Fast Persistent Key-Value Store," 2015. [Online]. Available: http://google-opensource.blogspot.com.es/2011/07/leveldb-fast-persistent-key-value-store.html

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[21] "Cap'n'Proto," 2015. [Online]. Available: https://capnproto.org/index.html

[22] "MongoDB ObjectId," 2015. [Online]. Available: http://docs.mongodb.org/manual/reference/object-id/

[23] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 4, pp. 481–494, Nov. 2006.

[24] "Apache Hadoop," 2015. [Online]. Available: https://hadoop.apache.org

[25] "Apache Hadoop and Apache HBase," 2015. [Online]. Available: https://www.safaribooksonline.com/library/view/using-flume/9781491905326/ch01.html

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, New York, NY, USA, 2003, pp. 29–43.

[27] C. Ellis, "Distributed data structures: A case study," *Computers, IEEE Transactions on*, vol. C-34, no. 12, pp. 1178–1185, Dec 1985.

[28] 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 Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[29] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur, "Pvfs: A parallel file system for linux clusters," in *In Proceedings of the 4th Annual Linux Showcase and Conference*. MIT Press, 2000, pp. 391–430.

[30] "PanFS," 2015. [Online]. Available: http://www.panasas.com/products/panfs

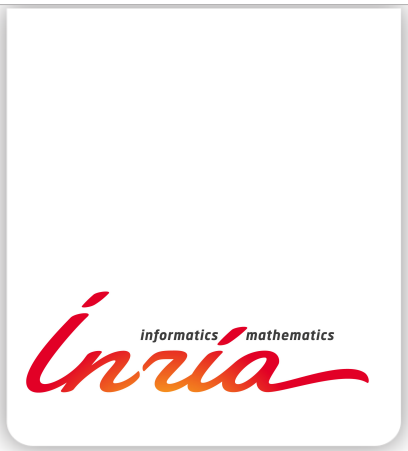
[31] "Giraffa File System," 2015. [Online]. Available: https://github.com/GiraffaFS/giraffa

[32] "Apache HBase," 2015. [Online]. Available: https://hbase.apache.org

[33] "Lustre," 2015. [Online]. Available: http://lustre.org/

[34] A. Thomson and D. J. Abadi, "CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 1–14.

Contents



RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803