Optimizing Distributed In-memory Storage Systems: Fault-tolerance, Performance, Energy Efficiency

Mohammed Yacine Taleb

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

école normale supérieure rennes

THESE DE DOCTORAT DE

L'ECOLE NORMALE
SUPERIEURE RENNES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

« Yacine TALEB »

« Optimizing Distributed In-memory Storage Systems: Fault-tolerance, Performance, Energy Efficiency»

Thèse présentée et soutenue à « Inria Rennes Bretagne Atlantique», le « 02 Octobre 2018»
Unité de recherche : Inria Rennes Bretagne Atlantique

Rapporteurs avant soutenance :

Maria S. Perez Professeure, Universidad Politécnica de Madrid, Espagne
Pierre Sens Professeur, Sorbonne Université, France

Composition du Jury :

Président :	Guillaume Pierre	Professeur, Université de Rennes 1
Examinateurs :	Christian Perez	Directeur de recherche, Inria Grenoble – Rhône-Alpes
	Maria S. Perez	Professeure, Universidad Politécnica de Madrid, Espagne
	Pierre Sens	Professeur, Sorbonne Université, France
Dir. de thèse :	Gabriel Antoniu	Directeur de Recherche, Inria Rennes Bretagne Atlantique
Co-dir. de thèse :	Toni Cortes	Maître de Conférences, Universitat Politècnica de Catalunya, Barcelone

Invité

Ryan Stutsman	Maître de Conférences, University of Utah, Salt Lake City

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

école normale supérieure rennes

Titre : Optimisation des Systèmes de Stockage en Mémoire : Performance, Efficacité, Durabilité

Mots clés : Stockage en mémoire, Performance, Efficacité, Durabilité

Résumé : Les technologies émergentes, telles que les objets connectés et les réseaux sociaux sont entrain de changer notre manière d'interagir avec autrui. De par leur large adoption, ces technologies génèrent de plus en plus de données. Alors que la gestion de larges volumes de données fut l'un des sujets majeurs de la dernière décennie, un nouveau défi est apparu récemment : comment tirer profit de données générées en temps réel. Avec la croissance des capacités de mémoires vives, plusieurs fournisseurs services, tel que Facebook, déploient des péta-octets de DRAM afin de garantir un temps d'accès rapide aux données. Néanmoins, les mémoires vives sont volatiles, et nécessitent souvent des mécanismes de tolérance aux pannes coûteux en terme de performance.

Ceci crée des compromis entre la performance, la tolérance aux pannes et l'efficacité dans les systèmes de stockage basés sur les mémoires vives. Dans cette thèse, nous commençons, d'une part, par étudier ces compromis : nous identifions les facteurs principaux qui impactent la performance, l'efficacité et la tolérance aux pannes dans les systèmes de stockage en mémoire. Ensuite, nous concevons et implémentons un nouveau mécanisme de réplication basé sur l'accès à la mémoire distante (RDMA). Enfin, nous portons cette technique à un nouveau type de système de stockage : les systèmes de stockage pour streaming. Nous concevons et implémentons des mécanismes de réplication et de tolérance aux pannes efficaces et un impact minimal sur les performances sur le stockage pour streaming.

Title : Optimizing Distributed In-memory Storage Systems: Fault-tolerance, Performance, Energy Efficiency

Keywords : In-memory Storage, Performance, Efficiency, Durability

Abstract. Emerging technologies such as connected devices and social networking applications are shaping the way we live, work, and interact with each other. These technologies generate increasingly high volumes of data. Dealing with large volumes of data has been an important focus in the last decade, however, today the challenge has shifted from data volume to velocity: How to store, process, and extract value from data generated With the growing capacity of DRAM, service providers largely rely on DRAM-based storage systemsto serve their workloads. Because DRAM is volatile, usually, distributed in-memory storage systems rely on expensive durability mechanisms to persist data.

This creates trade-offs between performance, durability and efficiency in in-memory storage systems We first study these trade-offs by means of experimental study. We extract the main factors that impact performance and efficiency in in-memory storage systems. Then, we design and implement a new RDMA-based replication mechanism that greatly improves replication efficiency in in-memory storage systems. Finally, we leverage our techniques and apply them to stream storage systems. We design and implement high-performance replication mechanisms for stream storage, while guaranteeing linearizability and durability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Historically, data has been used to drive businesses but today it is becoming crucial to all aspects of human life. Emerging technologies such as smartphones, autonomous cars, smart home devices, and personal assistants are shaping the way we live, work, and interact with each other. These technologies generate increasingly high volumes of data [27, 50].

Applications such as social networks or online shopping, e.g. Facebook and Amazon, are driving these changes. They offer services that can be easily accessed by a large number of users. For instance, Facebook reports 2 billion active users and 8 billion video uploads per day [32].

Such technologies generate tremendous volumes of data. Dealing with large volumes of data has been an important focus in the last decade, however, today the challenge has shifted from data volume to velocity: How to store, process, and extract value from data generated in real-time?

From the storage point of view, traditional disk-based storage systems have long been present in the ecosystem of web and Big Data applications. However, they could not sustain the level of performance demanded by the emerging data-intensive applications. With the growing capacity of DRAM, service providers largely rely on DRAM-based storage systems to serve their workloads. For instance, caching has been widely adopted by service providers to serve frequently accessed data. More recently, a new class of in-memory storage systems aim to fully capture DRAM performance by providing availability and durability, therefore, eliminating the need for a another layer of persistence.

Because DRAM is volatile, usually, distributed in-memory storage systems persist data on secondary storage such as disk. Therefore, they have specific mechanisms to guarantee

high availability, especially in case of failures when they must read data from slow persistent storage. This has two implications: (1) in-memory systems have to create redundant data copies (typically with replication) during normal operations, which incurs additional overhead; (2) to achieve high availability and quickly recover from failures, many systems leverage mechanisms that use resources aggressively. For instance, the RAMCloud storage system requires all machines in a cluster to participate during crash recovery [61].

As DRAM is still expensive compared to other storage media, system designers carefully tuned performance, focusing less on other metrics, such as energy consumption, or CPU efficiency, i.e., how many CPU cycles spent on each operation. While solid-state storage and network speeds witnessed tremendous evolution in speed and capacity, CPUs frequencies haven't followed the pace, hit by physical limits. Moore's law stated that the number of transistors in a chip doubles every two years. The rate held steady from 1975 to 2012. For the last 5 years, CPU clock speeds have capped at 4 GHz [16]. Because the CPU is intensively involved in I/O operations, it is becoming the main bottleneck in the data path.

As a result, it appears that operating DRAM-based storage systems will:(1) incur high operating costs in terms of energy consumption and resource utilization; (2) it is less appealing to collocate applications with DRAM-based storage, which is, by definition, one of the main use cases of in-memory storage systems [61, 21, 59].

While lots of research efforts have been dedicated to optimizing the performance of in-memory storage, we decided to focus on fault-tolerance. In this thesis, we show that there is a wide gap to fill to make in-memory storage systems more efficient by first studying their performance and energy efficiency. We show that durability mechanisms, such as replication, can be quite inefficient. To solve these issues, we propose a new replication mechanism, called *Tailwind*, by leveraging hardware trends and new networking technologies. Finally, we take advantage of our experience in developing Tailwind to design and implement durability in *Kera*, a low-latency stream-storage system. We show that by leveraging state-of-the-art hardware technologies coupled with lessons learned from our contributions, it is possible to bring high performance, strong consistency, and durability altogether to stream-storage systems, which is considered to be challenging [38, 28, 65].

This thesis was carried-out in the context of the BigStorage ETN with the goal of characterizing fault-tolerance overheads in in-memory storage systems and optimize these mechanisms to enable better performance and more efficient fault tolerance mechanisms.

1.1 Contributions

The main contributions of this thesis can be summerzied as follows:

Investigating the Efficiency of In-memory Storage Systems

Most large popular web applications, like Facebook and Twitter, keep most of their data in the main memory. While prior work has focused on how to exploit the low-latency of in-memory access at scale, there is very little visibility into the efficiency of in-memory storage systems. For instance, no prior work investigated the cost of operations in terms of CPU cycles or energy consumption. Even though it is known that main memory is a fundamental energy bottleneck in computing systems (i.e., DRAM consumes up to 40% of a server's

power). In this contribution, by the means of experimental evaluation, we have studied the performance and efficiency of RAMCloud — a well-known in-memory storage system. We focus on metrics such as performance, CPU cycles per operations, and energy consumption. We reveal non-proportional power consumption due to the polling mechanism. We also find that the current replication scheme implemented in RAMCloud limits the performance and results in high energy consumption. We show that replication can play a negative role in crash recovery in some configurations.

Tailwind: Fast and Atomic RDMA-based Replication

Replication is essential for fault-tolerance. However, in in-memory systems, it is a source of high overhead as we have shown in the first contribution. Remote direct memory access (RDMA) is an attractive means to create redundant copies of data, since it is low-latency and has no CPU overhead at the target. However, existing approaches still result in redundant data copying and active receivers. In state-of-the-art approaches, receivers ensure atomic data transfer by checking and applying only fully received messages. We propose Tailwind, a zero-copy recovery-log replication protocol for scale-out in-memory databases. Tailwind is the first replication protocol that eliminates *all* CPU-driven data copying and fully bypasses target server CPUs, thus leaving backups idle. Tailwind ensures all writes are atomic by leveraging a protocol that detects incomplete RDMA transfers. Tailwind substantially improves replication throughput and response latency compared with conventional RPC-based replication. In symmetric systems where servers both serve requests and act as replicas, Tailwind also improves normal-case throughput by freeing server CPU resources for request processing. We implemented and evaluated Tailwind on the RAMCloud in-memory key-value store. Experiments show that Tailwind improves RAMCloud's normal-case request processing throughput by 1.7×. It also cuts down writes median and 99th percentile latencies by 2 and 3 respectively.

Bringing High Performance, Durability, and Linearizability to Stream Storage

Data-intensive applications such as IoT, self-driving cars, analytics, and scientific simulations show unprecedented needs in terms of storage performance. They exhibit similar requirements: a stream-based data model, low-latency and high-throughput, and in many cases they require exactly-once processing semantics. However, existing solutions, such as Apache Kafka, struggle when it comes to high availability and strong consistency, and do not perform very well. In this contribution we present the design and implementation of durability mechanisms in Kera, a low-latency stream-storage system. Kera provides fine grain data access, allowing high-throughput, while scaling to the load of clients. Moreover, Kera enables continuous data service even in the event of failures without sacrificing linearizability.

1.2 Publications

Papers in International Conferences

- **Yacine Taleb**, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. *Tailwind: Fast and Atomic RDMA-based Replication*. In Proceedings of the 2018 USENIX Annual Technical

Conference, (**USENIX ATC'18**). July 2018, Boston, United States.

- **Yacine Taleb**, Shadi Ibrahim, Gabriel Antoniu, and Toni Cortes. *Characterizing Performance and Energy-efficiency of The RAMCloud Storage System*. In Proceedings of the The 37th IEEE International Conference on Distributed Computing Systems (**ICDCS'17**). May 2017, Atlanta, United States.

Workshops and Demos at International Conferences

- **Yacine Taleb**. *Optimizing Fault-Tolerance in In-memory Storage Systems*, in the EuroSys 2018 Doctoral Workshop (**EuroDW'18**). April 2018, Porto, Portugal.
- **Yacine Taleb**, Shadi Ibrahim, Gabriel Antoniu, Toni Cortes. *An Empirical Evaluation of How The Network Impacts The Performance and Energy Efficiency in RAMCloud*, Workshop on the Integration of Extreme Scale Computing and Big Data Management and Analytics in conjunction with IEEE/ACM **CCGRID'17**. May 2017, Madrid, Spain.
- **Y. Taleb**, S. Ibrahim, G. Antoniu., T. Cortes. *Understanding how the network impacts performance and energy-efficiency in the RAMCloud storage system*. Big Data Analytics: Challenges and Opportunities, held in conjuction with ACM/IEEE **SC'16**. November 2016, Salt Lake City, United States.

1.3 Organization of the Manuscript

The rest of this manuscript is organized as follows.

We discuss traditional storage systems and their limitations in Chapter 2. Then, we give a brief overview of the main hardware trends that lead to the emergence of in-memory storage systems. Next, we introduce in-memory storage systems and describe the main issues related to efficient-resource usage, energy efficiency, and the cost of fault-tolerance mechanisms in these systems.

Chapter 3 presents an in-depth experimental study. It characterizes the performance and energy efficiency of the RAMCloud storage system. We obtain many findings that helped us design an efficient replication protocol.

Chapter 4 presents Tailwind, an RDMA-based replication protocol. Tailwind is the first protocol that leaves backups idle during during replication. A key enabler for Tailwind is its ability to rebuild metadata knowledge in case of failures. We present its design and implementation in RAMCloud. Finally, we show how Tailwind can substantially increase performance while reducing the CPU footprint.

In Chapter 5, we focus on durability and availability in Kera, a new stream-storage system. We describe the design of the replication and crash recovery mechanisms in Kera. In contrast with existing systems, we demonstrate that by leveraging modern networking technologies and careful design, it is possible to provide performance, exactly-once semantics, and availability in stream storage. Finally, we provide an evaluation of Kera against state-of-the-art stream-storage systems.

Chapter 6 concludes this thesis, summarizes contributions, and gives future perspectives.

Chapter 2

In-Memory Storage Systems: Background and Challenges

Contents

To set the context of our work, this chapter first describes the evolution of Big Data applications and what challenges they raise for storage systems. Then, we take a closer look

at traditional large-scale storage systems and explain why they are unfit to satisfy the needs of emerging applications. After a brief description of new hardware trends, we describe the advent of in-memory storage systems and their prominent role in supporting current and future Big Data applications. Then we explain the importance of durability and the associated challenges in the context of in-memory storage systems. Finally, we present the motivation that drove our research and the current thesis.

2.1 Data-Intensive Applications

Data is driving our lives. Most of the services and interests we use are data-driven: sales, entertainment, health, sports, etc. Easy-to-use products are becoming pervasive, for instance messaging, gps, e-commerce, entertainment. Facebook, Instragram, and WhatsApp count more than a billion active users each on their messaging services [32]. Naturally, this translates in an unprecedented growth in the data volumes generated. But it also means data comes faster and in different forms. In [80], Stonebraker introduces Big Data according to the "3Vs" model. Big Data refers to datasets that are of a big volume, need big velocity, or exhibit big variety.

Volume. Nowadays, data is generated at an unprecedented rate. This is largely due to Internet-based companies such as Facebook or Google. For instance, Google is estimated to manage 15 exabytes of data (i.e., 15 billion gigabytes of data) in 2015 [27]. Contributing to this huge amount of data, Youtube is roughly storing 1 exabytes of videos. This volume is expected to increase up to 3 exabytes in 2025 [95]. The volume is particularly challenging when organizations have to process data.

Variety. Data is generated and collected in many formats. From structured data stored in databases to text and binary objects. However, more recently, SAS [74] estimates that 80% of organization data is not numerical. Regardless of its format, data needs to be stored and efficiently processed which can be challenging.

Velocity. This is probably the most challenging aspect of Big Data nowadays. Beside the large Volume and high Variety of data, Big Data processing systems usually have to handle data at a very fast arrival rate. For instance Facebook receives more than 350 millions photos daily [50]. Steam, an online gaming platform, has to handle 17 million players connected simultaneously while providing less than 250 milliseconds response times [78]. Therefore, performing analytics on data arriving at such rates requires a storage system that has an equivalently fast data access speed.

Let us have a look at some representative data-intensive applications. We present the characteristics of these applications and their impact from the storage infrastructure perspective. We discuss as well the energy-efficiency challenge raised when designing and deploying data-intensive storage infrastructures.

2.1.1 Internet-Based Applications

Internet-based services and applications such as social networks and mobile apps had probably the fastest growing number of users in the last decade. They also exhibit various needs in terms of performance, consistency, and durability.

Facebook. Facebook has more than a billion active users who record their relationships, share their interests, upload text, images, and video, and curate semantic information about their data [9]. A key enabler for personalized experience comes from **timely**, **efficient**, and **scalable** access to this flood of data. Facebook relies on a specialized storage system to provide billions of reads per second and millions of writes per second [9].

Netflix. With almost 100 million active users, Netflix data-pipeline captures roughly 500 billion events per day, which translates to approximately 1.3 petabyte of data per day. At peak hours, it can record 8 million events per second [4] and 24 gigabyte per second. In order to handle these huge flows of data and provide **real-time** recommendations to users, Netflix relies on 150 clusters of state-of-the-art stream-storage systems.

Banking and Finance. Goldman Sachs Investement has more than $1.4 trillion in assets under management. To make the best investment decisions, they rely on real-time analytics and have more than 1.5 terabyte of data traffic per week [91] from various sources. Although it is not as huge as social apps, they have very restrictive requirements: there can be **no data loss**, even in the event of a data-center outage; **no data inconsistencies**; and no **failover** scenarios. Similarly, ING Bank manages more than 1 billion incoming message per day [85] from their clients. While handling this large flow of data, they have to provide **uninterrupted** and **consistent** data access 24/7.

2.1.2 Scientific Applications

Large Hardon Collider (LHC). CERN's LHC is the world's largest and most powerful particle collider, the most complex experimental facility ever built and the largest single machine in the world. The aim of the LHC is to allow physicists to test the predictions of different theories of particle physics, including measuring the properties and search for new physical particles. Experiments from LHC generates about 30 petabytes of data per year [83]. In order to deal with such **large volumes of data**, the LHC federates a large computing grid community comprising over 170 facilities. However, data generated from experiments is expected to grow to 400 petabytes in a decade.

Square Kilometer Array (SKA). *The SKA Telescope* is an undergoing project federating 11 countries, which will address some of the key problems of astrophysics and cosmology [77]. The goal is to build a a set of telescopes that can take very high resolution pictures of the space in order to understand the origin of the universe and evolution of cosmic magnetism [77]. SKA is very representative of the storage challenges that we will face in the near future. Each telescope is SKA will write data at a rate of 1 terabyte per second and read at a rate of 10 terabytes per second [77]. While the software storage of SKA will have to handle **extreme data transfer rates**, it has to provide **durability** and **crash recovery** as well [77].

2.1.3 Emerging Applications Requirements

With ever growing data volumes and velocity, there is a need for low-latency and high-throughput access to data. For instance, Facebook social graph has to sustain a throughput of billions of reads and writes per second [9]. More recently, new stream-processing applications require sub-millisecond processing time. Some use cases from our everyday lives require fast processing: they include fraud detection, national security, stock market management, and autonomous cars. For instance, operating autonomous cars requires a latency of less than 50 ms while ensuring high reliability [75].

Storage plays probably one of the most crucial roles in operating these applications. Storage has to meet all above-cited requirements such as fast and consistent-data access. However, with recent concerns about datacenters energy consumption, system designers must also account for the energy efficiency. This is even more difficult when considering the fact that performance and energy efficiency are two opposing goals.

2.2 Storage Systems: Evolution and Challenges

Applications have different needs that are related to data access. However, system designers and service providers not only try to meet applications needs, they have additional challenges to deal with. For instance, energy efficiency is a first-order concern in the design of datacenters. The cost of powering server systems has been steadily rising with higher performing systems, while the cost of hardware has remained relatively stable [26]. Some studies even report that the cost of the energy consumed by a server during its lifetime would exceed the cost of the equipment itself [6]. As an example, Google, which spent 500 $million in a recent datacenter [90], would have to forecast the same cost just for the energy bills.

Besides the economic incentive, there are alarming ecological concerns. In 2012, Google used as much electricity as Turkey to operate its datacenters [84]. Because a major fraction of that electricity is produced from coal-driven power stations [84], operating datacenters has a considerable and direct impact on the environment.

While it is implicit, storage systems play a big role in the overall energy efficiency of an infrastructure, and more precisely, how *efficiently* that infrastructure is used. We refer to the efficiency of a storage system as the number of CPU cycles per operation performed. Different operations can have different values, for instance a write operation in a system will naturally spend more CPU cycles than a read. This metric is tightly related to the energy-efficiency of storage systems, which we define as *Watts* consumed per operation performed. Implicitly, improving the efficiency of a storage system (by reducing the number of CPU cycles) will directly reduce its energy footprint.

For instance, the SKA project is estimated to have a power consumption of 100 MW, which will absorb two-thirds of the total cost of the project over 50 years [77]. Savings in energy cost translate very directly into increased capacity for user support and reduced costs. However, one of the main challenges is to have software that can operate the infrastructure under a certain budget [26], while still satisfying required levels of performance.

Next we present a high-level overview of most relevant design aspects of storage systems

.

Performance. In a context where most of applications are data-intensive, the rate at which data can be written or read from the storage system is of great importance, therefore, performance usually impacts most of the design of storage systems.

Scalability. This property consists in keeping the same level of performance when the number of clients accessing the system grows. Another aspect of scalability is the ability to perform better when increasing the resources used to run the system.

Fault tolerance. Large scale system are usually composed of a large number of components interacting with each other, at such scale faults are unavoidable. Therefore, it is important for a system to be able to recover from fault transparently, i.e., without degrading too much performance or sacrificing other features.

Consistency. Since a large number of clients interact with storage systems, there can be concurrent accesses to the same data. Depending on the requirements, storage systems can offer different guarantees regarding data access semantics. Concretely, strong guarantees mean that, from the client's perspective, operations on data always happen as if there were no other clients in the system. In contrast, weaker semantics mean that a system can tolerate some arbitrary order among some operations, reducing synchronization costs.

Efficiency. The efficiency of storage systems (i.e., CPU cycles per request processed) and their energy efficiency have a linear relationship(see § 2.2). While the efficiency is less regarded compared to performance or consistency, it still plays an important role in systems design. There are many techniques to improve the efficiency and save energy in storage systems, ranging from low-level approaches such as Dynamic Voltage Frequency Scaling (DVFS) [10], to shutting down a set of machines in a cluster to save energy.

2.2.1 A Brief Overview of Traditional Storage Systems

In this section we discuss the main design approaches that impacted modern storage systems. We present the main features of every type of system, their limitations, and finally we introduce the NoSQL family of storage systems.

SQL-Database Management Systems (DBMS) Databases store collections of data, i.e., schema, tables, queries, reports, views, and other elements. Databases had and continue to have a large success due to their well-defined data model and query language (SQL) [18]. DBMS are designed around the ACID (Atomicity, Consistency, Isolation, Durability) properties, which intend to guarantee the validity of transactions in the event of faults (failures, power outages, etc.). As a result, it is easy, from the point of view of the user, to use and even build applications on top of such systems since they guarantee data integrity while offering a rich query language on top.

File systems Similarly to DBMS, file systems have been invented in to provide an interface for end users and applications to store data in a structured manner. With a hierarchical organization, where directories contain files (or sub-directories), the main objective of file systems was to manage the location of files. However, with increased data volumes, the number of files increases and limits the scalability of the system, often due to the lack of scalability of the associated metadata management.

Parallel file systems were introduced in order to overcome centralized file systems scalability and failure tolerance limitations. They rely on decentralized storage that enables scalable performance and fast access. By spreading data and workloads across multiple servers, parallel file systems can provide scalable performance. Moreover, by replicating data and metadata, parallel file system can survive failures and thus provide durability.

Notification-Based Systems Supporting analytics over data which is constantly "moving" (called *stream processing*) requires a different kind of storage abstraction and particular communication protocol. Message-Oriented Middlewares or Brokers have been proposed to this purpose [17]. Message brokers allow two kind of models: *queuing* and *publish-subscribe*. Queuing is a point-to-point communication model whereby a pool of consumers may read from a server and each message is delivered to one of them. Publish-subscribe is a broadcast communication model whereby a message is broadcasted to all consumers [81].

2.2.2 Limitations of Traditional Storage Systems

However, "traditional" storage systems can no longer sustain new applications needs. The main reason is that very often most of the features we defined in § 2.2 are opposite. The best illustration is the *CAP* theorem. Eric Brewer introduced this well-know theorem stating that at most only two out of the three following properties can be achieved simultaneously within a distributed storage system: **C**onsistency, **A**vailability and **P**artition Tolerance [8]. Many applications depend on these features to sustain good user quality of experience. Before discussing the implications of the CAP theorem, let us introduce the three features:

Consistency. The consistency guarantees that a reader will receive the most recent value of a data object.

Availability. The availability refer to the ability of the system to sustain continuous service, i.e., every client operation must be processed and a response must be issued. An alternative definition given by some Cloud vendors is to guarantee 99.99 % of the requests are served under certain Service Level Objectives (SLO).

Partition Tolerance. Tolerating partitions means the system is able to operate even in the event of network disruption between replicas. The system must be able to keep consistency and availability even if it is partitioned.

The CAP theorem had a lot of impact on distributed systems design and implementation over the last 15 years [8]. Specifically, for geo-replicated systems, partition tolerance is still a major issue. For instance if a system replicated across two regions gets partitioned, there will clearly be a divergence between data on the two partitions, if the systems keeps processing

client requests. The Wide-Area Network (WAN) latency plays a major role in that. Therefore, systems have to sacrifice either consistency or availability in order to tolerate partitions.

2.2.3 NoSQL Datastores

The NoSQL movement appeared after the CAP theorem was introduced. A concrete example of failure of some of the traditional storage systems concerns DBMS. the In the 1990s, most database management experts believed that a single general-purpose DBMS could meet substantially all needs [60]. Their idea is that by adding enough data types and access methods they could meet almost any requirement. However, over the last decade RDBMS have shown their scalability limits to face Big Data overwhelming. This is mainly because ACID semantics are hard to provide at large scale. Growing applications scales pushed system designers to sacrifice part or all of the ACID guarantees at the expense of scalability and availability. Alternatively to the ACID model that imposes a lot of restrictions, the BASE model [66] was proposed:

Basically Available. This is not availability in the sense of CAP, i.e., there will be a response to *any* request. For instance, one could imagine that every request will receive a response within 100ms. But, this does not guarantee that the correct response will be given, for example a "failure" could be returned.

Soft State. The state of the system could change over time, even when there are no client requests. For instance, The system can queue requests, (e.g. writes) and differ their execution to favor availability.

Eventually Consistent. The system will eventually become consistent once clients stop sending requests. Once data will propagate to all nodes, the state of the state of the system should stabilize.

Using the BASE model, many systems were designed and implemented [13, 20, 45, 57], principally *trading consistency for availability*. However, there is an increasing number of applications and use-cases that need consistency, availability, and performance altogether. Examples of these applications include self-driving cars, fraud detection, banking, etc. Recently, researchers and system designers have started exploring different paths and system designs based on recent hardware.

2.3 In-memory Storage Systems

2.3.1 Hardware Trends Impact on In-memory Storage Systems

Interestingly, recent hardware improvements have shaken things in storage systems design. Some recent projects have demonstrated that it is possible to provide strong consistency, high availability, and tolerating faults at the same time [21, 61]. New technologies are enabling levels of performance that were unthinkable just several years ago. Among these changes, probably the most prominent one is the shift of storage from disk to memory. For decades storage systems were primarily using disk as the main storage medium. Moving storage

to main memory could improve the performance up to 1000x for some applications [61]. Naturally, this opens new perspectives and invalidates many restrictions for building fast and consistent distributed storage systems.

2.3.1.1 DRAM and I/O Devices Evolution

With the growing popularity of Internet, especially with the advent of the world-wide web, commodity hardware appeared as a viable alternative to high-end machines. Both academia and industry realized that commodity hardware can provide decent performance at reduced cost. This trend continued, especially with the advent of Big Data applications and more precisely with the Map Reduce big data processing model. MapReduce is a programming model which targets efficient data processing across large-scale distributed infrastructures. It leverages the divide-and-conquer technique [19] in order to distribute the large amount of work across distributed clusters of commodity machines.

In the past, high-speed networks and large-memory machines were only available on high-end machines. However, in the last two decades, commodity hardware improved in all aspects and at a fast pace. For instance, the price per DRAM bit has dropped by 1000x from 1998 to 2018 [53]. Similarly, Ethernet based-networks have evolved from Gigabit Ethernet in years 2000 to a 400GbE and Terabit Ethernet recently [24].

This has dramatically changed the view of system designers regarding hardware resources and bottlenecks. For instance, HDFS, which was designed to run on commodity-hardware, stores all data on cheap disks and only keeps very frequently accessed data in memory. However, nowadays system designers realized that it is possible to fit all working data sets in the memory of commodity machines [96]. This has lead to the design of systems such as Spark [96].

2.3.1.2 Low-Level Features

While hardware advances played an important role in the advent of in-memory storage, there are some software features that were equally important for efficient-data acccess.

Kernel Bypass, Zero Copy. Over the years, OS network stacks have become a bottleneck to fast network processing. Network speeds are evolving at such a rate that, for instance, various mechanisms (such as interprocessor interrupts, data copies, context switches, etc) all add overheads [7]. Moreover, relying on a fully-fledged TCP/IP stack further introduces overhead due to additional data copying from user space to kernel space (transmit buffers). In order to cope with CPU frequencies slow improvement, a recent trend is to perform *kernel bypass* and *zero copy* [43], i.e., perform network packet processing outside of the OS kernel. By doing so, an application can see its latency reduced by up to a factor of 5x [7].

Remote Direct Memory Access (RDMA). Remote Direct Memory Access (RDMA) is a networking protocol that provides direct memory access from a host node to the memory of remote nodes, and vice versa. RDMA achieves its high bandwidth and low latency with no CPU overhead by using zero-copy transfer and kernel bypass . There are several RDMA implementations, including InfiniBand, RDMA over Converged Ethernet (RoCE) [71], and iWARP [31].

Over the years, these features helped in mitigating well-known bottlenecks like slow disk I/O access. Moreover, they play an important role in leveraging DRAM speed. For instance, without RDMA or kernel-bypass, DRAM-based storage system would be bottlenecked by network, reducing or even eliminating the attractiveness of using DRAM to store data.

2.3.2 State-of-the-art of In-memory Storage Systems

Thanks to all these features described above, in-memory storage systems appeared as a solution to provide low-latency access to very large volumes of data. The term *In-memory Storage* rallies a large set of storage systems that rely on main memory for data storage. In-memory storage systems are faster than disk-optimized storage systems because disk access is slower than memory access, the internal optimization algorithms are simpler and execute fewer CPU instructions.

2.3.2.1 Databases

SAP HANA *SAP HANA* [29] is an in-memory, column-oriented, relational database management system. Besided storing its data in memory and serving as a database server, it performs advanced analytics (predictive analytics, spatial data processing, text analytics, text search, streaming analytics, graph data processing) and includes ETL capabilities as well as an application server. Storing data in main memory rather than on disk provides faster data access and, by extension, faster querying and processing [30]. HANA does not store all of its data in memory but rather applies a technique called "Dynamic Tierring", i.e., identifying "hot" data and placing it in memory, while putting "cold" data on disk.

H-Store The *H-Store* system [41] is a highly distributed, row-store based relational database that runs on a cluster on shared nothing, main memory executor nodes. Every node in an H-Store deployment is comprised of a set "sites" that represent the basic operational entities in the system. A site is a single-threaded daemon that an external OnLine Transaction Processing (OLTP) application connects to in order to execute a transaction. Each site is independent from all other sites, and thus does not share any data structures or memory with collocated sites running on the same machine.

2.3.2.2 File Systems

Tachyon. *Tachyon* is an in-memory distributed and fault-tolerant file system for Big Data processing framework [48]. The main idea behind Tachyon is to avoid replication and use lineage (§ 2.3.3) to achieve fault-tolerance Tachyon. Lost output is recovered by re-executing the operations (tasks) that created the output. As a result, lineage provides fault-tolerance without the need for replicating the data [48]. Tachyon addresses the lineage re-computation overhead by analyzing the data and the tree that lead to its creation. By doing so, Tachyon can checkpoint data in a clever way, bounding the recomputation time during recovery.

Apache Ignite. The Apache Ignite File System (IGFS) is an in-memory file system delivering similar functionality of Hadoop HDFS [76] but only in memory. IGFS splits the data from each file into separate data blocks and stores them in a distributed in-memory cache.

However, unlike HDFS, IGFS does not need a name node and automatically determines file data locality using a hashing function. IGFS can be transparently plugged into Big Data processing frameworks such as Hadoop.

2.3.2.3 Distributed Caches

Caching is a well-known technique to store data or the result of a computation in a hardware component that can be rapidly accessed compared to improve future I/Os. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests can be served from the cache, the faster the system performs.

In-memory caches played an important role since the earliest days of operating systems. For instance, early versions of UNIX used caches to improve the performance of the file system. More recently, with the advent of large-scale web-based applications caches became an important component in service providers architecture. In this situation, caches are usually used to speed up database accesses.

Memcached. *Memcached* [52] is a general-purpose distributed memory caching system. Memcached's APIs provide a very large hash table distributed across multiple machines. When the table is full, subsequent inserts cause older data to be purged in least recently used (LRU) order. Recently, Facebook presented how it scaled Memcached to make it a distributed, replicated key-value cache [59]. It is mainly used for database query cache but also as a general purpose cache to store complex computations such as Machine Learning (ML) results. A notable improvement made is the ability to replicate data within a single datacenter and across geographically distributed regions for load balancing.

2.3.2.4 Persistent In-memory Key-Value Stores

Using memory as a cache has many advantages. It is easy to use, deploy, and offers relatively good performance. However, data stored in caches has to be persisted in a different (slower) storage layer. This approach forces developers to manage consistency between the cache and the backing store, and its performance is limited by cache misses and backing store overheads. Given the increasing DRAM size and its decreasing cost, many systems appeared providing general-purpose facilities for accessing data in DRAM. This approach has many advantages:

- Preventing developers from managing consistency between the cache and the backing store.
- Eliminating backing store overheads in terms of communication, persistence, and data duplication.
- Unlocking full DRAM speed by keeping every byte of data in memory.

RAMCloud. RAMCloud [61], a general-purpose distributed storage system that keeps all data in DRAM at all times. RAMCloud combines three overall attributes: low latency, large scale, and durability. For that, RAMCloud requires state-of-the-art networking technologies. It offers access latencies as low as 4 µs. RAMCloud aggregates all of their memories into a single coherent key-value store which allows large storage capacities. Although RAMCloud keeps all data in DRAM, it also maintains backup copies of data on secondary storage to ensure a high level of durability and availability. This frees application developers from the need to manage a separate durable storage system, or to maintain consistency between in-memory and durable storage.

Redis. Redis [69] is an open-source in-memory distributed, in-memory key-value store with optional durability. Redis supports a rich data model, such as strings, lists, maps, sets, etc. Redis typically holds the whole dataset in memory. Persistence is achieved by "snapshotting"; the dataset is asynchronously transferred from memory to disk from time to time. Redis supports master-slave replication. A slave may be a master to another slave. This allows Redis to implement a single-rooted replication tree. Redis slaves can be configured to accept writes, permitting intentional and unintentional inconsistency between instances. Note that replication is useful for read (but not write) scalability or data redundancy.

By default, Redis writes data to a file system at least every 2 seconds, with more or less robust options available if needed. In the case of a complete system failure on default settings, only a few seconds of data would be lost.

FaRM. FaRM [21] is a main memory distributed computing platform that exploits RDMA to improve both latency and throughput compared to main memory systems that use TCP/IP. FaRM exposes the memory of machines in the cluster as a shared address space. Applications can use transactions to allocate, read, write, and free objects in the address space with location transparency. A key enabler for FaRM performance is a fast message passing interface implemented on top of RDMA. Similarly to RAMCloud, FaRM achieves availability and durability using replicated logging to SSDs.

2.3.3 Durability and Availability in In-memory Storage Systems

As we explained previously, to fully exploit DRAM's potential, in-memory storage systems have to provide durability and availability [61]. Durability means that data is persisted in some way and can be recovered in case of faults. Availability means that the system must always be able to handle requests, even during failures. These two features are important to ensure high performance and continuous service for clients. However, there are many ways of providing durability and availability, and each one of them has advantages and constraints.

2.3.3.1 Redundancy

Redundancy is probably the most well-known way of providing durability and availability. In a distributed environment, it consists of duplicating data to multiple machines. By doing so, if one or multiple machines fail, or if data corruption happens, the storage system can reconstruct a copy the original data by leveraging duplicates.

Erasure Coding. Erasure codes, also known as forward error correction (FEC) codes, create a mathematical function to describe a set of numbers so they can be checked for accuracy and recovered if one is lost. In a simple mathematical form, erasure coding consists of the following equation: $n = k + m$, where k is the original amount of data and m stands for the extra or redundant symbols that are added to provide protection from failures. By doing so, the original data can be regenerated out of any n chunks. This way the system can support the loss of up to k chunks.

In storage systems, to implement erasure coding, data is considered as a collection of fixed-size chunks. Every k chunks are encoded into $n - k$ additional equal-size coded chunks (called parity chunks). Reed-Solomon (RS) codes are one well-known example of erasure codes [98].

In in-memory storage systems, erasure coding is attractive because of its low space overhead. For instance, some systems report up to 50% memory savings compared to replication [98]. However, the downside is that erasure coding needs additional processing compared to replication which can dramatically increase CPU usage [98].

Replication. Data replication is a widely used approach to provide redundancy. A commonly used technique is Primary-backup Replication (PBR), where each primary node has M backups to store its data replicas and can tolerate M stop failures. One of the backup nodes would act as the new primary node if the primary node failed, resulting in a view change. Other replication techniques exist, for instance, Active Replication (AR), where all replicas receive and process the clients requests. Therefore, all replicas need to apply data changes deterministically. However, it is rather costly in practise, especially in systems providing strong consistency, where PBR is preferred.

2.3.3.2 Lineage.

Lineage has become increasingly popular with the advent of Big Data processing frameworks. Lineage does not store redundant data, but rather "how" the data was generated. In lineage, lost output is recovered by re-executing the jobs that created the output. A concrete example of lineage is the Resilient Distributed Datasets (RDDs) [96]. Formally, an RDD is a read-only, partitioned collection of records. RDDs do not need to be materialized at all times. Instead, an RDD has enough information about how it was derived from other datasets (its lineage) to compute its partitions from data in stable storage. This is a powerful property: in essence, a program cannot reference an RDD that it cannot reconstruct after a failure.

Lineage is attractive for in-memory cluster computing logging and replication would interfere with the processing frameworks by consuming extra CPU cycles and network bandwidth. In contrast, RDDs provide an interface based on coarse-grained transformations (e.g., map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its lineage) rather than the actual data [96].

2.3.4 Discussion

All these techniques come with trade-offs. For instance, erasure coding induces a lot of computation overhead. Therefore, it does not fit most design goals of low-latency storage systems. On the other hand, lineage can only be applied to specific systems such as

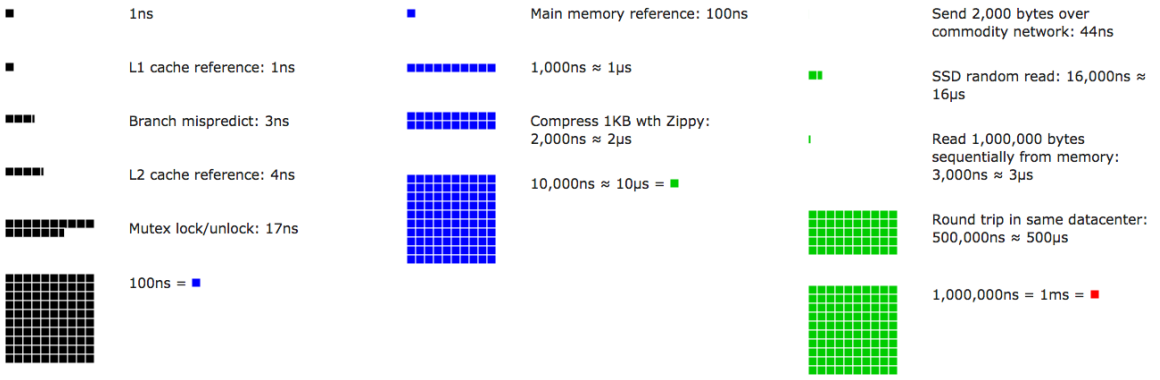


Figure 2.1 – Different I/O latencies as of 2018.

in-memory cluster computing. Moreover, lineage does not offer the same availability guarantees as replication: in case of failures, re-computing the output of data might take very long time. This results in a trade-off between the frequency of checkpointing lineage and re-computation time.

Replication is the only general purpose technique that can be used in most of today's in-memory key-value stores. However, even so, we argue that replication is expensive, especially in the context of in-memory storage systems. Next we discuss the trade-offs between performance, energy efficiency, and durability/availability in persistent in-memory key-value stores.

2.4 Current Challenges of In-memory Systems

Persistent in-memory storage systems undeniably paved a path to building fast, consistent, and fault-tolerant storage systems. By relying on cutting edge hardware, they can achieve unprecedented levels of performance. A key enabler for that is the ability to extensively use or monopolize resources (exclusive access). For instance, RAMCloud has a unique architecture allowing it to provide ultra-low latency. However, it always needs to dedicate a core for polling network requests to achieve low latency. Similarly, FaRM builds a fast message passing interface on top of RDMA. Yet, it uses all available CPUs to poll requests [21] to enable high throughput and low latency. Many other in-memory storage systems have similar mechanisms or exhibit the same intensive resource usage [40, 49, 61, 21].

2.4.1 Efficient Resource Usage

As shown in figure 2.1 the memory is orders of magnitude faster than SSDs and HDDs. Therefore, to fully exploit such potential, researchers and system designers came up with techniques that can leverage highly performing new hardware, though, at the cost of efficiency. As a result, many systems *waste resources* in order to provide *performance*, *consistency*, and *fault-tolerance*.

Moreover, as explained in § 2.3.1.1, almost all I/O devices have witnessed a considerable performance bump. However, CPU frequencies scaling has not witnessed the same evolu-

tion. Moore's law stated that the number of transistors in a chip doubles every two years. The rate held steady from 1975 to 2012. For the last 5 years, CPU clock speeds have capped at 4 GHz [16]. Clocks speeds have also suffered due to miniaturization, higher clock speeds mean more energy use and therefore more heat buildup.

New technologies such parallel I/O bring a totally new dimension to the demand for more cores within the CPU architecture [14]. Despite a notable bump in the number of parallel CPU cores, it's not easy to fully exploit such architectures. First, there is coordination overhead between the cores to ensure cache coherency. Today, a two- or four-processor machine is not really two or four times as fast as a single CPU even for multi-threaded applications [15]. Second, unless all cores are running different processes, or different threads of a single process that will run completely independently, without having to wait for each other, they won't be fully utilized.

Recent studies have shown that among NoSQL datastores, a majority exhibit high CPU overheads [62]. The main reason is that they have been designed with the assumption that CPU is the fastest component of the system. However, even some in-memory storage systems like Cassandra, require several tens or hundreds of thousands of cycles per operation. For relatively small data items it would therefore need several modern cores to saturate a single 1 Gbit/s link [62]. Furthermore, a recent study has unveiled how Memcached and Redis poorly leverage the CPU architecture [97]. For instance, for small objects, both systems exhibit a high instruction cache miss rate leading to useless CPU utilization while instructions were fetched from memory.

2.4.2 Energy Efficiency

There are increasing concerns about datacenters energy consumption. For instance, they will cost $13B to the American businesses by 2020 [23]. A plethora of studies exist on improving the energy efficiency of storage systems. They basically allow servers to be turned off without affecting data availability.

For example, GreenHDFS [42] separates the HDFS cluster into hot and cold zones. The new or high-access data are placed in the hot zone. Servers in the cold zone are transitioned to the power-saving mode and data are not replicated. On the other hand, Rabbit [2], introduce an energy-efficient distributed file system that maintains a primary replica on a small subset of always-on nodes (active nodes). Similarly, Sierra [87], introduce a new power-proportional distributed storage system which guarantees data availability when turning servers off thanks to its replication scheme.

While these optimizations concern disk-based storage systems, it is not yet clear how keeping all data in main memory would impact the energy efficiency. Some studies reported that DRAM-based main memories consume from 25% to 40% of a server's total power consumption [88].

In this thesis, we address this issue by providing a comprehensive experimental study on the performance and energy efficiency of in-memory storage systems. By studying the state-of-the-art in-memory key-value store RAMCloud, we show the energy impact of low-latency architectures in in-memory key-value stores. More importantly, we show that durability and availability techniques can negatively impact the performance and energy efficiency of in-memory key-value stores. This contribution is detailed in Chapter 3.

2.4.3 Availability and Durability

Replication is fundamental to fault tolerance, but at the same time they are costly. In some systems, backup servers don't process user-facing requests, but in many systems each node acts as both a primary for some data items and as a backup for other data items. In some systems this is implicit: for example, a key-value store may store its state on HDFS.

Replication is expensive for three reasons. First, it is inherently redundant and, hence, brings overhead: the act of replication itself requires moving data over the network. Second, replication in strongly consistent systems is usually synchronous, so a primary must stall while holding resources while waiting for acknowledgements from backups (often spinning a CPU core in low-latency stores). Third, in systems, where servers (either explicitly or implicitly) serve both client-facing requests and replication operations, those operations contend.

To cope with these limitations, we propose *Tailwind*, a new RDMA-based replication protocol. Tailwind leaves backups CPU completely idle during replication. While RDMA introduces consistency issues, Tailwind provides backups with a recovery protocol that guarantees data consistency and integrity in case of failures. We develop this contribution in Chapter 4.

2.4.4 A Use Case: Low-latency and Durable Storage For Streaming

Nowadays, Applications that require real-time processing of high-volume data steams are pushing the limits of traditional data processing infrastructures. For instance, these stream-based applications include financial, military, fraud detection, and various sensor-based services.

Recently, there is a need for applications that deal with a large amount of data arriving continuously that also need to generate an **accurate** analysis of that data in the face of late arriving data, data arriving out of order and failure conditions [65]. This means that the system ingesting data from various sources, has to provide **durability**, and **availability** while guaranteeing **strong consistency** at the same time. However, most of existing systems fail to meet all these requirements together. The main reason is that availability and durability are expensive. For instance, the team developing Apache Kafka reported that the whole team spent 9 months designing the exactly-once processing semantics in Kafka [44]. Activating this feature in Kafka reduces by *at-least* 20% the throughput of the system [38].

By leveraging lessons learned from our previous contributions, we designed efficient replication and fault tolerance mechanisms for *Kera*, a new stream-storage system. We show that by leveraging state-of-the-art networking technologies and DRAM-based storage, it is possible to provide exactly-once semantics and high performance at the same time. This contribution is developed in Chapter 5.

2.5 Summary: Optimizing In-memory Storage Systems

Providing **fast** access to **accurate** data is probably the biggest challenge storage systems are facing nowadays. For example, The Square Kilometre Array (SKA) project expects ingestion rates of 1 Terabyte/s [77]. Data will continue to arrive at extreme rates, especially with the advent of IoT applications.

On the other hand, recent hardware trends have made it possible to build systems that can sustain unprecedented data access rates. However, most of these usually sacrifice efficiency at the expense of performance. Emerging applications such as IoT, self-driving cars, and web applications need: high **efficiency** under various workloads, e.g. writes such as graph applications; they require predictable **low latency**, as for many parallel computing frameworks, the total execution time is dominated by the slowest task; they need **atomic** data access due to the large number of concurrent access they exhibit; **durability** and **availability** are crucial, for instance exactly-once stream processing cannot be achieved without these two features.

In a context where energy is one of the major concerns, there is an urgent need for more efficient system designs. In this thesis, we argue that it is possible to significantly improve the efficiency of in-memory storage systems by providing more efficient fault-tolerance mechanisms.

To summarize,

- There is a lack of efficient design for in-memory storage systems.
- There is little visibility in the energy efficiency of in-memory storage systems; as memory is become one of the main storage mediums, it becomes critical to identify potential sources of inefficiency.
- Durability and availability are usually expensive; By definition they require redundancy and data movement which puts a lot of pressure on scarce resources such as CPUs.

In the next chapters, we characterize the performance and energy efficiency of in-memory storage to expose sources of inefficiency (chapter 3). We then propose solutions to efficiently provide durability and availability (chapter 4). Finally, we show that it is possible to have provide performance with durability and availability thanks to more efficient techniques for fault-tolerance (chapter 5).

Chapter 3

Investigating the Efficiency of In-memory Storage Systems

Contents

3.1 Energy Efficiency vs. Performance

In-memory storage systems have became a key building block for service providers and large scale applications. They enable fast access to large volumes of data even under highly concurrent accesses. To meet emerging application needs, recent in-memory key-value stores also provide features such as data durability, scalability, and memory efficiency.

However, service providers have also to design for the efficiency of their infrastructure, which sometimes can be contradictory with application requirements. For instance, energy efficiency is a first-order concern in the design of datacenters. Yet, most of energy-saving techniques in disk-based storage systems negatively impact performance. Understanding the trade-offs between performance and energy efficiency can lead to better system designs that conciliate both.

DRAM-based storage is fundamentally different from disk-based storage. There is little, to no visibility on the efficiency or energy consumption of in-memory storage systems because their architecture is quite different from traditional storage systems. Moreover, DRAM is known to be quite energy hungry as some studies reported that DRAM-based main memories consume from 25% up to 40% of a server's total power consumption [88].

The goal of this chapter is to shed the light on the interplay between performance and energy consumption in in-memory storage. We carry out an experimental study on the Grid'5000 [1] testbed, and use the RAMCloud in-memory key-value store. We design and run various scenarios that help us to identify the main performance bottlenecks and sources of energy-inefficiency. More precisely we answer the following questions:

- *What is the energy footprint of peak performance of RAMCloud?* We reveal issues of power non-proportionality that appear with read-only applications.
- *What is the performance and energy footprint of read-update workloads?* We focus on these workloads (mixed reads and updates) as they are prevalent in large scale Web applications [5].
- *How does RAMCloud's replication scheme impact performance and energy?* Though it is supposed to be transparent in RAMCloud, we find that replication can be a major performance and energy bottleneck.
- *What is the overhead of RAMCloud's crash-recovery in terms of availability and energy consumption?* In addition we study how replication affect these metrics. Surprisingly, we show that replication can play a negative role in crash-recovery.
- *What can be improved in RAMCloud, and for in-memory storage?* We discuss possible improvements for RAMCloud and derive some optimizations that could be applied in in-memory storage systems.

It is important to note that, although RAMCloud is used as an experimental platform, our target is more general. Our findings can serve as a basis to understand the behavior of other in-memory storage systems sharing the same features and provide guidelines to design energy-efficient in-memory storage systems.

3.2 A Representative System: RAMCloud

Ideally, the main attributes that in-memory storage systems should provide are performance, durability, availability, scalability, efficient memory usage, and energy efficiency. Most of today's systems target performance and memory efficiency [54, 25, 49]. Durability and availability are also important as they free up application developers from having to backup in-memory data in secondary storage and handle the synchronization between the two levels of storage. On the other hand, scalability is vital, especially with today's high-demand Web applications. They are accessed by millions of clients in parallel. Therefore, large scale clustered storage became a natural choice to cope with such high demand [52].

In contrast with most of the recent in-memory storage systems, RAMCloud main claims are performance (low-latency), durability, scalability, and memory efficiency. The other closest system to provide all these features to be found in the literature is FaRM [21]. Unfortunately it is neither open-source nor publicly available.

3.3 The RAMCloud Storage System

The rest of this section describes the high-level details of the RAMCloud storage system. It provides an overview of the system that we use as a basis for the rest of the thesis.

3.3.1 Data Model

RAMCloud is a general purpose key-value store. Data in RAMCloud is divided into tables, each of which is identified by a unique textual name and a unique 64-bit identifier. A table may contain any number of objects. Objects include the following information: a *key*, a *value*, and a *version number*.

3.3.2 Architecture

A RAMCloud cluster, as depicted in Figure 3.1, consists of three entities: a coordinator maintaining meta-data information about storage servers, backup servers, and data location; a set of storage servers that expose their DRAM as storage space; and backups that store replicas of data in their DRAM temporarily and spill it to disk asynchronously.

Each storage server comprises two modules. A master component handles data storage and operations such as reads and writes from clients. A backup component uses its persistent storage, either disk or flash, to persist data that belongs to master on *other* servers.

3.3.3 Log-based Storage

A master uses an append-only log-structured memory to store its data and a hash-table to index it. The log-structured memory of each server is divided into 8MB segments. The segment granularity is a trade-off between many factors, but it was chosen primarily to achieve high I/O throughput.

The other major data structure on a master is a hash table, which contains an entry for each object. The hash table allows a master to quickly find objects that match a given table and key.

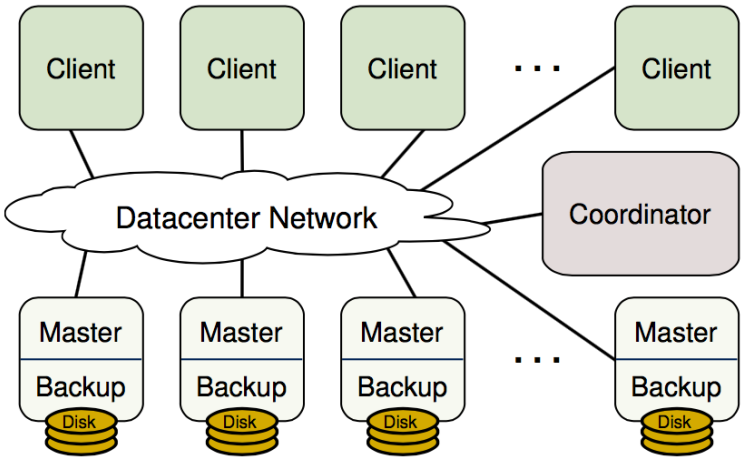


Figure 3.1 – Each storage server contains two modules: a master, which manages objects in its DRAM, and a backup, which stores segment replicas on disk or flash. A central coordinator manages the cluster of masters and backups. Client applications access RAMCloud with remote procedure calls [73].

A master replicates its log to a configurable number of replicas to achieve durability. For each segment, a master will randomly choose a replica. Randomness allows a master to spread its segments uniformly across the cluster, thus reducing data loss probability and increasing recovery performance in case of crashes.

3.3.4 Write Operations

When a master receives a write request from a client, it appends a new entry for the object to its head log segment, creates a hash table entry for the object (or updates an existing entry), and then replicates the log entry synchronously in parallel to the backups storing the head segment. During replication, each backup appends the entry to a replica of the head segment buffered in its memory and initiates an I/O operation to write the new data to secondary storage. it responds to the master without waiting for the I/O to complete. When the master has received replies from all backups, it responds to the client. The buffer space on each backup is freed once the segment has been closed (meaning that a new head segment has been chosen and this segment is now immutable) and the buffer contents have been written to secondary storage [61]. This approach has two attractive properties. First, writes complete without waiting for I/O to secondary storage. Second, backups use secondary storage bandwidth efficiently: under heavy write load, they will aggregate many small objects into a single large block for I/O [61].

3.3.5 Cleaning

Whenever a master deletes or modifies an object, it appends a tombstone record to the log, which indicates that the previous version of the object is no longer valid. Over time, free space will accumulate in the logs as objects are deleted or overwritten, therefore, a master has to reclaim unused storage.

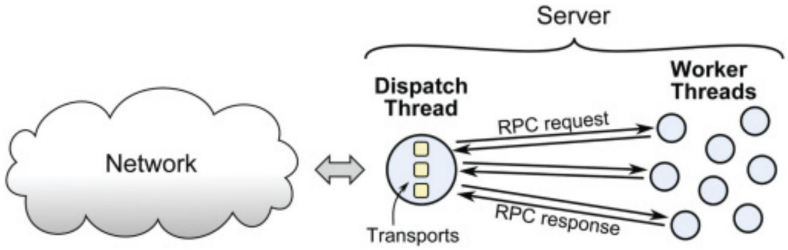


Figure 3.2 – A single dispatch thread handles all network communication; it passes each incoming RPC request to a worker thread for handling [61].

Each RAMCloud master and backup has a clearner module. The cleaner scans over all segments, identifies old or deleted objects, and reclaims free space. It works by allocating seperate segments, copying live data to them, thus ignoring stale objects. Survivor segments are merged in batch to avoid contention with regular write operations. Note that the cleaner starts its work at high memory utilization, which happen to be the best policy as reported in [73].

There is also a cleaner running at each backup. Although it runs less frequently, since disk storage is cheaper than DRAM, it reflects the work performed by the cleaner in the primary to the backup.

3.3.6 RPC Subsystem

RAMCloud relies on a low-latency RPC architecture in order to achieve high performance. Although we present only design aspects allowing low latency RPCs, the subsystem also handles failures, retries, and all aspects related to integrity check.

Architecture. RAMCloud threading architecture is depicted in 3.2. To achieve the lowest possible latency, RAMCloud dedicates pins a thread to a core to dispatch requests as fast as possible. The rest of the threads are called, worker threads, they execute regular code such as reads, writes, cleaning, etc. Modules involved in moving RPC requests, i.e. Transports, implement a polling mechanism. The dispatcher thread will continuously invoke those polling methods, for instance specific transports such as the Infiniband-based transport, will continuously check for incoming network requests.

The lifetime of an RPC is typically as follows: when the dispatch thread detects an incoming request, it will select an idle worker thread and assigns that specific RPC to it; (2) when the worker finishes its work it stores a pointer to the RPC and becomes idle; (3) the dispatch thread hands the RPC to the appropriate transport for transmission.

Kernel Bypass. RAMCloud heavily relies on kernel bypass to achieve low latency. NIC devices register memory space in the application address space instead of the kernel allowing applications to directly communicate with the NIC. Applications send packets to the NIC by specifying virtual addresses, which means NICs have to have a physical-to-virtual memory mapping. Kernel bypass dramatically improves performance, however it is not a feature available in all software and hardware stacks today. Nevertheless it is becoming more and more adopted in modern datacenter networking [61].

3.3.7 Fault tolerance

Fault tolerance is one of the main and most complex features in RAMCloud [61]. Since RAMCloud has low latency goals, it must recover from failures as fast as possible without disturbing normal operations. RAMCloud assumes a stop-failure model.

Crash Recovery. Because RAMCloud adopts the Primary-backup Replication (PBR) scheme, it cannot continue normal operation even if a single copy is lost . Therefore, if a server crashes, the data that was stored on that specific server becomes unavailable until recovery is completed. As a result, crash recovery impacts availability. In order to achieve fast crash recovery, RAMCloud tries to leverage as many backups as possible during recovery. Actually, it is crash recovery that impacted how replicas are selected during write operations. A second principle to achieve fast crash recovery, it to leverage recovery parallelism; instead of recovery a master data into a single other master, RAMCloud shards the crashed master data into multiple partitions and assigns each partition to a single recovery master. By doing so, RAMCloud can recovery a 64 GB master in less than 2 seconds [61].

3.4 Experimental Study

In order to answer the questions stated in § 3.1, we decided to conduct an extensive experimental campaign. We designed a set of experiments associated to specific features. In the following, we first describe the metrics we used for our evaluation, then we describe the platform we used to deploy RAMCloud, along with the generic configuration.

3.4.1 Metrics

We took as main metrics the throughput of the system (i.e., the number of requests served per second) and the power consumption of nodes running RAMCloud. We used as well the energy efficiency metric [92], i.e., $Number\_of\_requests\_served/joule$.

3.4.2 Platform

The experiments were performed on the Grid'5000 [1] testbed. The Grid'5000 platform provides researchers with an infrastructure for large-scale experiments. It includes 9 geographical sites spread across French territory and one located in Luxembourg.

We used Nancy's site nodes to carry out our experiments. More specifically, the nodes we used have 1 CPU Intel xeon X3440, 4 cores/CPU, 16 GB RAM, and a 298 GB HDD. Additionally each node includes a Infiniband-20G and a Gigabit Ethernet cards. We have chosen these nodes as they offer capability to monitor power consumption: 40 of these nodes are equipped with Power Distribution Units (PDUs), which allow to retrieve power consumption through an SNMP request. Each PDU is mapped to a single machine, allowing fine grain power retrieval. We run a script on each machine which queries the power consumption value from its corresponding PDU every second. We start the script right before running the benchmark and stop it after all clients finish

3.4.3 RAMCloud configuration

Throughout all our experiments we make sure to reserve the whole cluster, which consists of 131 nodes, to avoid any interference with other users of the platform. We dedicate the 40 nodes equipped with PDUs to run RAMCloud's cluster, i.e., master and backup services. One node is used to run the coordinator service. The remaining 90 nodes are used as clients. We have fixed the memory used by a RAMCloud server to 10GB and the available disk space to 80GB. We have fixed the memory and disk to much larger sizes than the workloads used as we explain later in Section 3.4.6.

We configured RAMCloud with the option *ServerSpan* equal to the number of servers. As RAMCloud does not have a smart data distribution strategy, this option is used to manually decide how many servers each table will span. Data is then distributed uniformly.

3.4.4 Benchmark

We used the industry standard Yahoo! Cloud Serving Benchmark (YCSB) benchmarking framework [12]. YCSB is an open and extensible framework that allows to mimic real-world workloads such as large scale web applications, to benchmark key-value store systems. YCSB supports a large number of databases and key-value stores, which is convenient for comparing different systems.

To run a workload, one needs to fill the data-store first. It is possible to specify the request distribution, in our case we use uniform distribution. Executing the workload consists of running clients with a given workload specification. We used three basic workloads provided by YCSB: *Workload A* which is an update-heavy workload (50% reads, 50% updates), *Workload B* which is a read-heavy workload (95% reads, 5% updates), and *Workload C* which is a read-only workload. This combination of workloads has already been used in other systems evaluations [49, 98, 54].

When assessing RAMCloud peak performance as we do in Section 3.4.5, we use a workload of 5 M records of 1 KB, and 10 M requests per client. Running the benchmark consists of launching simultaneously one instance of a YCSB client on each client node. With 30 clients it corresponds to 300 M requests which represents 286GB of data requested per run. In some runs, e.g., when running 30 clients with a single RAMCloud node, the execution time reaches in average 4300 seconds, which we believe outputs enough and representative results to achieve our goals.

For the rest of the experiments (Sections 3.4.6 and 3.4.7, where we use update-workloads, we pre-load 100 K records of 1KB in the cluster. Each client issues 100 K requests, which corresponds to the total number of records. Having each client generate 100 K requests results in having 1 M requests with 10 clients for example, and 9 M requests with 90 clients, which corresponds to 8.58 GB of data requested per run. With workload A, it corresponds as well to 4.3 GB of data inserted per run. Therefore, we avoid saturating the main memory (and disk when using replication) of servers and trigger the cleaning mechanism.

In our figures, each value is an average of 5 runs with the corresponding error bars. When changing the cluster configuration (i.e., number of RAMCloud servers), we remove all the data stored on servers as well as in backups, then we restart all RAMCloud servers and backups in the cluster to avoid any interference with prior experiments. Overall, we made roughly 3000 runs with a total run time of approximately 1000 hours.

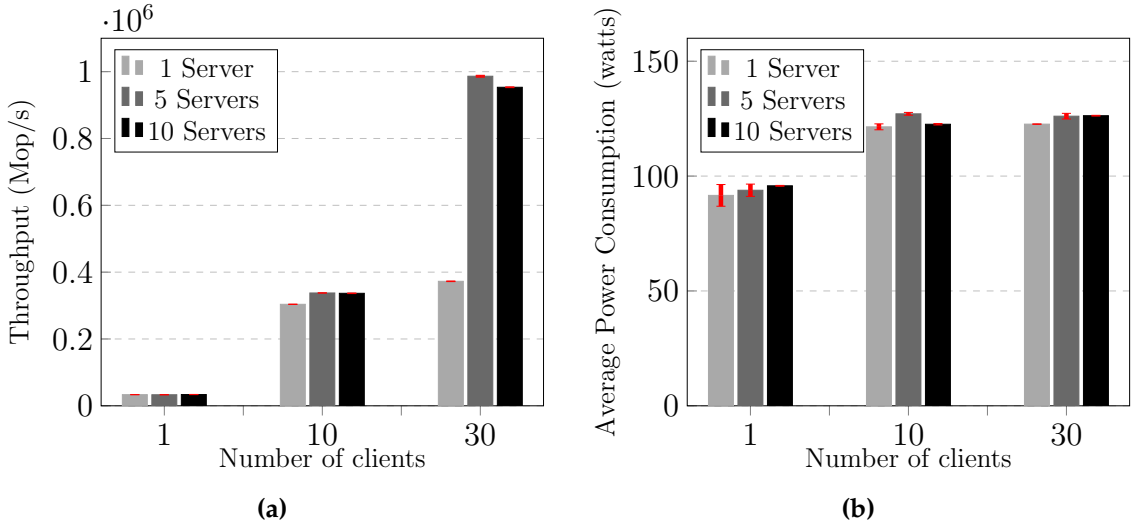


Figure 3.3 – The aggregated throughput (a) and average power consumption per server (b) as a factor of the cluster size

3.4.5 The Energy Footprint of Peak Performance

In this section we present our experiments on understanding of the maximum achievable performance. It is important since it gives us a landmark that can help us compare with our next experiments and identify potential performance bottlenecks. Furthermore, given that baseline we can compute energy efficiency as well.

3.4.5.1 Methodology

To allow RAMCloud to deliver its maximum performance, we configured our experiments as follows: (1) Disabling replication to avoid any communication between RAMCloud servers. In that way we have server-client communication only. RAMCloud issues heartbeat messages, but they are negligible compared to clients workload. (2) We use read-only workloads to avoid write-write race condition, and more importantly to prevent appending data to memory and triggering the cleaning mechanism of RAMCloud. (3) We use Infiniband network. (4) We make sure that the dataset is always smaller than memory capacity. (5) We distribute data uniformly (i.e., at RAMCloud cluster level) and requests (i.e., at the clients level) over the cluster to distribute work equally among servers and avoid hotspots. (6) We use a single client per machine to avoid any network interference at the NIC level of a machine or any CPU contention between client processes within a same machine.

We varied the RAMCloud cluster size from a single server to 5 and 10 servers. We did as well vary the clients number from one to 10 and 30 clients. We used the workload consisting of 5M objects and 10M read-only request per client.

The maximum throughput per server: Figure 3.3a shows the total aggregated throughput for this scenario. In one node experiment, the system scale up to 10 clients until it reaches its limit at 30 clients for a throughput of 372 Kreq/s (this is inline with the results presented in [73]). Increasing the number of servers to 5 improves the total throughput for 10 clients

and 30, achieving linear scalability. However, increasing the number of servers from 5 to 10 does not bring any improvement to the aggregated throughput achieved, which is due to the clients' limited rate at this point.

The corresponding power consumption: Figure 3.3b shows the average power consumption. With one server and a single client the average power is 92 W. It increases to 93 W and 95 W for 5 and 10 servers respectively. Even with smaller amount of work for 5 and 10 servers we can see that the power consumption remains at the same level. The same behaviour happens when increasing the load. For 10 clients the average power is between 122 W and 127 W. More surprisingly for 30 clients it stays at the same levels, i.e., 122 W and 127 W. This suggests that the servers are used at the same level (i.e., CPU) under different loads.

Non-proportional power consumption?: To confirm or invalidate this statement we looked into CPU usage of individual servers when running the workload. Table 3.1 displays the minimum and maximum CPU usage values. First we notice that the difference between the minimum and maximum values is not bigger than 2% for all scenarios. What is striking is to see that the difference between CPU usage of one node and 5 and 10 nodes is very small. The main reason is that RAMCloud hogs one core per machine for its polling mechanism, i.e., polling new requests from the NIC and delivering them to the application to achieve low-latency. Since we are using 4-core machines, this ends up using 25% of CPU all times, even without any clients.

Surprisingly, increasing the RAMCloud cluster size from 1 to 5 servers increases the aggregated throughput by 10% only while keeping the same CPU usage per node. We think this is an overhead to handle all concurrent incoming requests. By looking at the scenario of a single node, if we look carefully at the cases of 10 clients and 30 clients we can see a difference of 23% in throughput for a 1% difference in CPU usage. This suggests that this issue relates to the threads' handling. First, each additional running server has a dedicated polling-thread, which in our case translates to 25% of additional CPU usage. Moreover, not all threads are effectively utilized. For instance, the aggregated throughput and the average CPU usage per node are very similar for 1, 5, and 10 servers when servicing 10 clients. To mitigate the energy non-proportionality issue, one can think of carefully tuning the number of nodes in a cluster to meet the needs of the workloads.

Figure 3.4 shows the energy efficiency: As expected the energy efficiency is at its highest with a single server and with the largest number of clients. With 5 servers, the energy efficiency can barely reach half of the one of a single server. Further increasing the RAMCloud cluster size to 30 decreases the energy efficiency by a factor of 7.6x compared to the one of a single server.

Conclusion. RAMCloud is scalable in throughput for read-only applications. However, we find that it can have non-proportional power consumption, i.e., the system can deliver different values of throughput for the same power consumption. The reason is that servers reach their maximum CPU usage before reaching peak performance.

3.4.6 The energy footprint with read-update workloads

As demonstrated in [59] today's workloads are read-dominated workloads with a GET/SET ratio of 30:1. Hence, the questions we would like to answer are: How good is RAMCloud in

Clients \ Servers	1	5	10
	average	min | max	min | max
0	25	25 | 25	25 | 25
1	49,81	49,65 | 49,78	49,61 | 49,91
2	74,16	72,12 | 72,72	62,60 | 63,85
3	79,66	74,03 | 74,41	72,18 | 73,27
4	89,80	77,81 | 78,66	74,27 | 75,27
5	94,34	84,90 | 85,98	75,87 | 77,02
10	98,35	96,93 | 97,40	91,89 | 93,06
30	99,26	96,77 | 97,19	94,86 | 95,98

Table 3.1 – The minimum and maximum of the average CPU usages (in percentage) of individual nodes when running read-only workload on different RAMCloud cluster sizes and with different number of clients

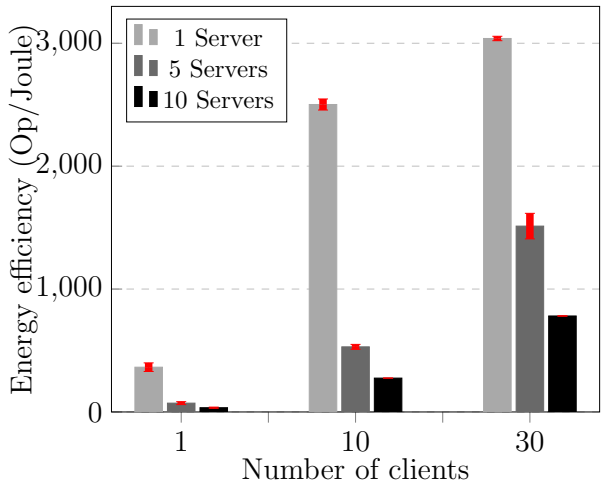


Figure 3.4 – The energy efficiency of different RAMCloud cluster sizes when running different number of clients.

terms of performance when running realistic workloads? Does it scale as well as it does for read-only workloads? How energy efficient is it when running realistic workloads? What are the architectural choices that can be improved?

3.4.6.1 Methodology

We used the same methodology as in § 3.4.5.1 except that we changed the workloads to have both read-update (95% reads, 5% updates) and update-heavy (50% reads, 50% updates) workloads. We recall that replication is disabled. It is important to note that we use different number of servers and clients compared to the previous experiment since the goal is different. Thereby, we do not compare the following scenario with the previous one. We use from 10 to 40 servers and from 10 to 90 client nodes. For space's sake we do not show all scenarios, but they all converge towards the same conclusions we give.

Clients \ Workload	A	B	C
	avg | err	**avg** | err	**avg** | err
10	**98K** | 4K	**236K** | 11K	**236K** | 18K
20	**106K** | 11K	**454K** | 11K	**482K** | 31K
30	**64K** | 4K	**622K** | 20K	**753K** | 16K
60	**63K** | 2K	**816K** | 26K	**1433K** | 38K
90	**64K** | 2K	**844K** | 19K	**2004K** | 31K

Table 3.2 – Total aggregated throughput (Kop/s) of 10 servers when running different with different numbers of clients. We used the three YCSB by default workloads A, B,and C

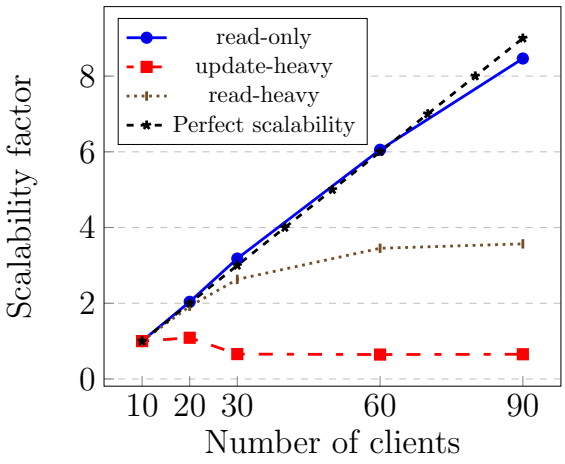


Figure 3.5 – Scalability of 10 RAMCloud servers in terms of throughput when varying the clients number. The "perfect" line refers to the expected throughput when increasing the number of clients. We take the baseline as the throughput achieved by 10 clients.

Comparing the performance with the three workloads: Table 3.2 shows the aggregated throughput for 10 servers when running the three workloads A, B, and C. For read-only workload the throughput increases linearly, reaching up to 2 Mop/s for 90 clients. For read-heavy workload the throughput increases linearly until 30 clients where it reaches 622 Kop/s. It increases in a much slower pace up to 844 Kop/s for 90 clients. The worst case is for update-heavy workload that surprisingly reaches its best throughput of 106 Kop/s when running 20 clients, then performance starts declining down to 64 Kop/s for 90 clients. At 90 clients, the throughput with workload C is 31x more than the one with heavy-update workload.

To have a better view on this phenomena, Figure 3.5 shows the ratio of the throughput when taking 10 clients as a baseline. We can see clearly that read-only applications have a perfect scalability, while read-heavy collapses between 30 and 60 clients. With heavy-update workload, throughput does not increase at all and degrades when increasing the number of clients.

Interestingly, we can see what impact do update operations have on performance. Since RAMCloud organizes its memory in a log-structured fashion writes should not have that

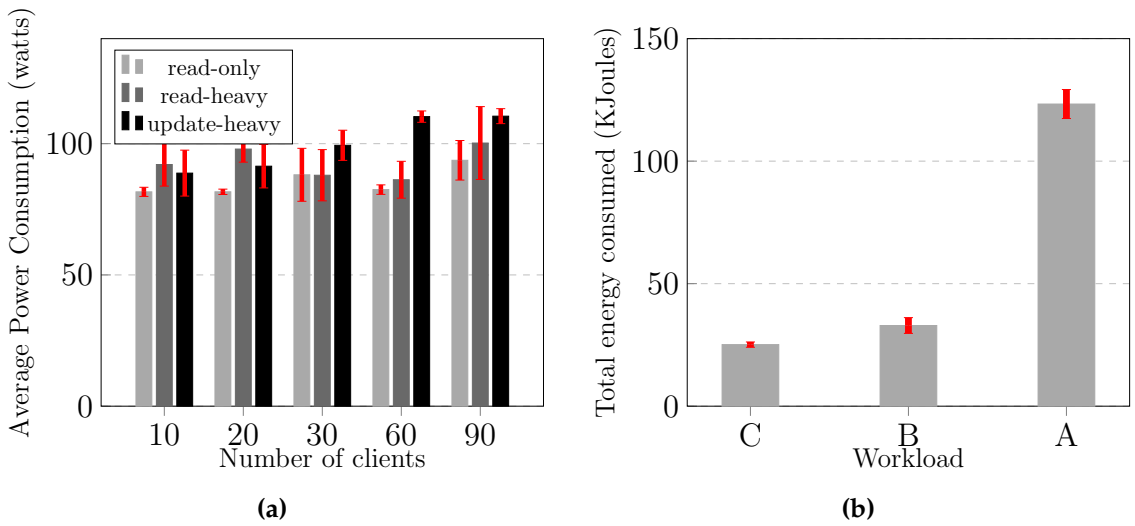


Figure 3.6 – (a) represents the average power consumption (Watts) per node of 20 RAMCloud servers as a function of the number of clients. (b) represents the total energy consumed by the same cluster for 90 clients as a function of the workload.

impact on performance, because every update/insert operation results in appending new data to the memory log and updating the hash-table. However, bringing up that much concurrency e.g., 90 clients leads to a lot of contention within a single server, and therefore servers will queue most of the incoming requests. This suggests that there is a poor thread handling at the server level when requests are queued.

More updates means more power per node?: The energy impact is straightforward when looking at Figure 3.6a that represents the average power consumption of the RAMCloud cluster. The power consumption per server when executing read-only workloads stays at the same level, i.e., 82 W up to 60 clients, after that it jumps to 93Watts. We can observe the same pattern for read-heavy except that it stands at a higher power consumption than read-only, which is around 92 W, then it goes up to 100 W for 90 clients. Power consumption of the heavy-update workload is the highest, though it starts with 10 and 20 clients around 90 W, it continues to grow with the number of clients to reach 110 W for 90 clients which is the highest value for all experiments.

To complement these results we show Figure 3.6b which displays the total energy consumed when running the three workloads with 90 clients. The total energy consumed is computed by multiplying the power consumption, of every server, collected every second by the execution time. We can see that read-heavy has 28% percent more energy consumption than read-only. The more surprising fact is the difference between heavy-update and read-only which is 492% more energy consumed for the heavy-update workload.

While it is expected that update operations take more time and processing than read operations, it is noteworthy to recall that RAMCloud relies on an append-only log-structured memory. By design updates are equivalent to inserts, therefore the additional overhead should mostly come from updating the in-memory hash-table and appending new data to the log. Consequently, we expected the gap between reads and updates to be very low as we have disabled replication in this particular case. We suspect the main cause being the

thread management at the server level. If all threads of a server are busy then upcoming requests will be queued, generating delays. Moreover, the number of servicing threads impacts performance as well. This number can depend on the hardware type (e.g., number of cores). Worse, it depends also on the workload type, e.g., in Figure 3.5 read-only scale perfectly while only update workloads experience performance degradation. For instance, performance will decrease under write-heavy workloads with more threads due to useless context switches, since all work could be done by a single thread. Whereas it is the opposite for read-only applications. Identifying this number empirically can bring substantial performance improvements. This issue was confirmed by RAMCloud developers [1].

Conclusion. We find that RAMCloud loses up to 57% in throughput with YCSB's read-heavy workload compared to read-only. With heavy-update workloads, the performance degradation can reach up to 97% at high concurrency (with replication disabled in both cases). Furthermore, we found that heavy-update workloads lead to a higher average power consumption per node, which can lead to 4.92x more total energy consumed compared to read-only workloads. We find this issue is tightly related to the number of threads servicing requests. Finding the optimal number can improve performance and energy efficiency, however, this is not trivial since it depends on the hardware and workload type.

3.4.7 Investigating Replication Impact

RAMCloud uses replication to provide durability and availability. There are two main techniques used to replicate data: *Primary-Backup* or *Symmetric* [98]. Since the per bit cost of memory is very high, primary-backup replication (PBR) is usually preferred to symmetric replication, especially when all data is kept in DRAM. RAMCloud uses PBR and keeps a single replica in memory to serve requests and pushes eventually replicas to disk as described [61]. Thus, the main concern about replication is at what extent it impacts the performance and energy consumption of the system and eventually look for possible improvements.

3.4.7.1 Methodology

We measure the transparency of the replication scheme by stressing it with an update-heavy workload (50% reads, 50% updates). This workload was preferred to update-only (100% updates) workload since the latter is far from what in-memory storage systems and Web storage systems were designed for [68]. We use the same parameters as in § 3.4.5.1, except the replication factor that we vary from 1 to 4. We change the number of RAMCloud servers from 10 to 40 and the clients from 10 to 60 nodes.

Replication impact on throughput: In Figure 3.7 we plot the total aggregated throughput when running heavy-update with different replication factors for 20 RAMCloud servers. When running with 10 clients we can see a clear decrease in throughput whenever the replication factor is increased, e.g., from replication factor 1 to 4 the throughput drops from 78 Kop/s to 43 Kop/s which corresponds to a 45% decrease. Further increasing the number of clients to 30 and 60 leads to an increase of throughput for replication factor 1 and 2 which

[1] https://ramcloud.atlassian.net/wiki/display/RAM/Nanoscheduling

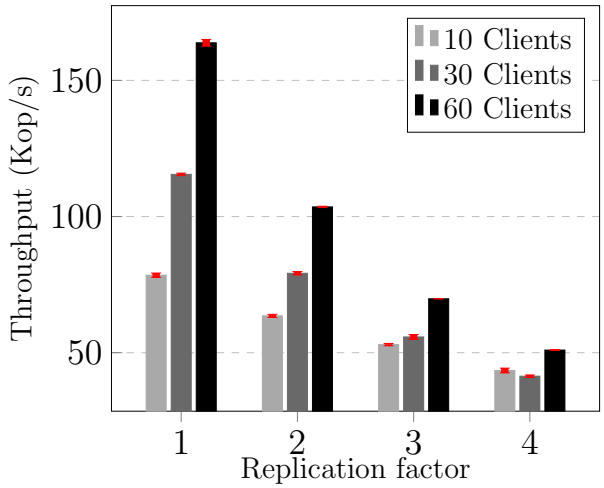


Figure 3.7 – The total aggregated throughput of 20 RAMCloud servers as a factor of the replication factor.

means that RAMCloud's cluster capacity has not been reached yet. Ultimately, for 30 and 60 clients setting the replication factor to 4 leads to a throughput of 41 Kop/s and 50 Kops, respectively, which means that we saturated RAMCloud cluster capacity.

While the impact of replication on performance might seem substantial, since the normal replication factor at which the system should run with is at least 3 or 4, the explanation lies on the replication scheme itself. In RAMCloud for each update request, a server will generate as many requests as its replication factor, e.g., if the replication factor is set to 4, upon receiving a request a server will generate 4 replication requests to other servers that have available backup service up and running. For every replication request a server sends, it has to wait for the acknowledgements from the backups before answering the client that issued the original request. This is crucial for providing strong consistency guarantees. Indeed, suppose a server can answer a client's request as soon as it has processed it internally and sent the replication requests without waiting for acknowledgements, then in case of a failure of that specific server, the RAMCloud cluster can end up with different values of the same data.

We plot in Figure 3.8a the aggregated throughput when running heavy-update at fixed rate of 60 clients. We varied the number of servers as well as the replication factor. It is interesting to see how enlarging the number of servers can reduce the load, e.g., when having replication factor set to 1 the throughput can be increased from 128 Kop/s to 237 Kop/s when scaling up the cluster from 10 to 40 servers. We remark the same behaviour for higher replication factors, though the throughput is lower when increasing the replication factor. It is noteworthy that the Figure does not include values for 10 servers when going beyond replication factor 2. The reason is that the experiments were always crashing despite the number of runs because of excessive timeouts in answering requests.

Replication impact on energy consumption: To give a general view on the power consumption under different replication factors we plot Figure 3.9 that shows the average power consumption per node of a cluster of 40 servers when running 60 clients for different replication factors. As expected the lowest average power is achieved with a replication factor of 1 is the lowest with an average power of 103Watts per server. Increasing the replication

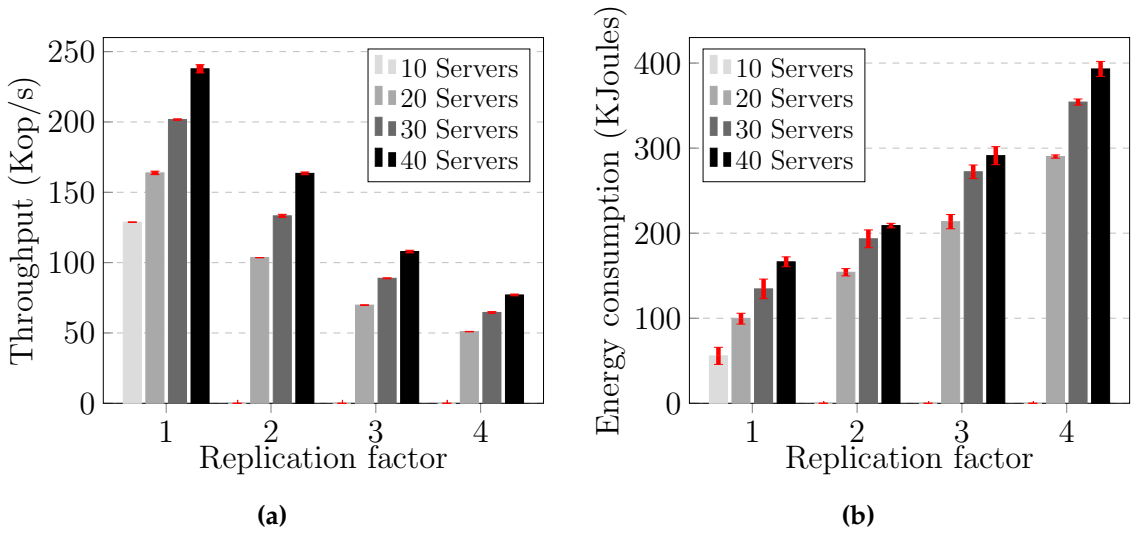


Figure 3.8 – (a) The total aggregated throughput as a function of the number of servers. The number of clients is fixed to 60. (b) The total energy consumption of different RAMCloud servers numbers as a function of the replication factor when running heavy-update with 60 clients.

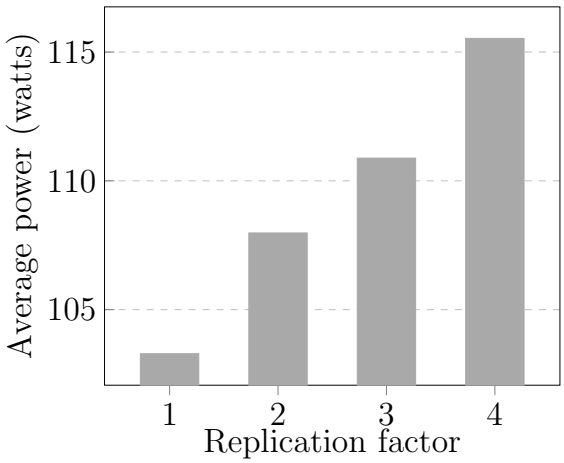


Figure 3.9 – The average power consumption per node of a cluster of 40 servers when fixing the number of clients to 60.

factor leads to an increase up to 115Watts per server for a replication factor of 4.

The rise in the power consumption per server results in a substantial increase in the total energy consumption of the RAMCloud cluster. Figure 3.8b illustrates the total energy consumption when fixing the number of clients to 60. Firstly, it is interesting to look at the difference in the total energy consumption when increasing the replication factor. For instance, with 20 RAMCloud servers and replication factor set to 1 the total energy consumed is 81 KJoules. It rises up to 285 KJoules which corresponds to an increase of 351%. For 40 servers the extra energy consumed reaches 345% whenever tuning the replication factor

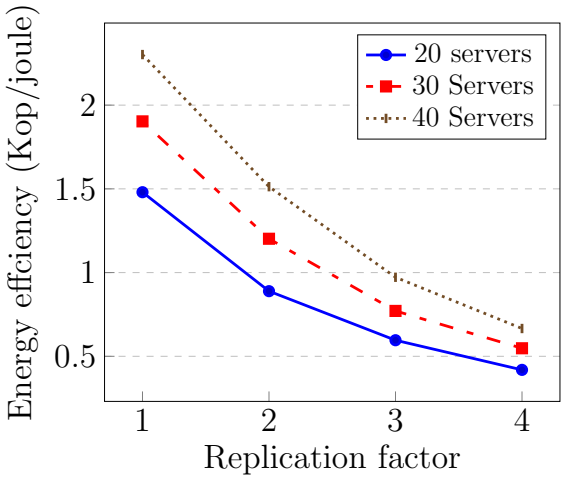


Figure 3.10 – The energy efficiency of different configurations as a function of the replication factor when running heavy-update with 60 clients

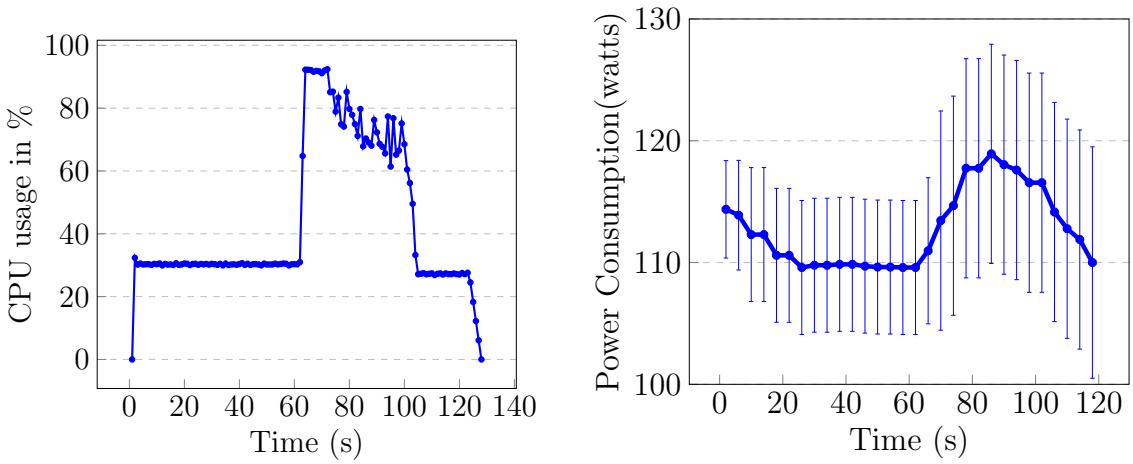
from 1 to 4. Secondly, it is noteworthy to compare the energy consumption when resizing the number of servers. As an example, with a replication factor of 1, the difference between the energy consumed by 20 and 30 servers is 16%. The increase in the energy consumed when scaling the cluster from 20 to 40 servers is 28%. When increasing the replication factor these ratios tend to decrease. More specifically, when the replication factor is fixed to 2, the difference in the total energy consumed by 20 and 30 servers is 10%, but this difference reaches 17% only when comparing 20 and 40 servers.

Conclusion. Increasing the replication factor in RAMCloud causes a huge performance degradation and more energy consumption. For instance, changing the replication factor from 1 to 4 leads to up to 68% throughput degradation for update-heavy workloads while it leads to in 3.5x additional total energy consumption. This overhead is due to the CPU contention between replication requests and normal requests at the server level. Waiting for acknowledgements from backups increases the overhead.

Changing the cluster's size. Figure 3.10 illustrates the energy efficiency when fixing the number of clients to 60. In a very surprising way, we can see that having more servers leads to better energy efficiency. As an illustration, when the replication factor is set to 1 the energy efficiency of 20 servers equals 1500 while for 30 servers it is 1900, finally, for 40 server it reaches 2300. The ratio tends to decrease whenever the replication factor is increasing.

It is noteworthy that the relative differences in the energy efficiency between different number of servers shrinks when increasing the replication factor. The explanation resides in Figure 3.8a where we can see that the relative differences in throughput decreases, therefore the fraction throughput/power goes down.

In contrast with the results from § 3.4.5 where the best energy efficiency for read-only workloads was achieved with the lowest number of servers, with heavy-update and replication enabled, it appears that provisioning more servers not only achieves better performance, but also leads to a better energy efficiency.



(a) **(b)**

Figure 3.11 – The average CPU usage (a) and power consumption (b) of 10 (idle) servers before, during, and after crash-recovery. At 60 seconds a random server is killed.

Conclusion. : Increasing the RAMCloud cluster size results in a better energy efficiency with update-heavy workloads when replication is active. Therefore, the type of workload should be strongly considered when scaling up/down the number of servers to achieve better energy efficiency.

3.4.8 Crash-recovery

Data availability is as critical as performance during normal operations since there is only a single primary replica in RAMCloud. Studying RAMCloud's crash recovery can help in understanding the factors that need to be considered when deploying a production system, e.g., replication factor, size of data per server, etc. Therefore, our main concern in this section is to answer the following questions: What is the overhead of crash-recovery? What are the main factors impacting this mechanism?

3.4.8.1 Methodology

Two important metrics we consider are recovery time and energy consumption. We perform the assessment by inserting data into a cluster and then killing a randomly picked server (after 60 seconds) to trigger crash-recovery, with different number of servers. First, we assess the overhead of a crash-recovery, and then we vary the replication factor to observe the impact on the recovery-time and the corresponding energy consumption.

The overhead of crash-recovery. For our first scenario we setup a RAMCloud cluster of 10 servers. We then inserted 10M records which corresponds to 9.7GB of data uniformly split across the servers, i.e., each server receives 1M records. It is noteworthy that we have set the replication factor to 4 which corresponds to the normal replication factor used in production systems [11].

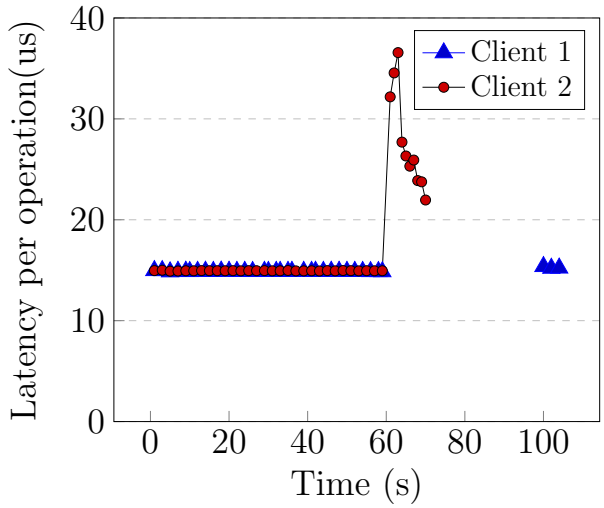


Figure 3.12 – The latency per operation before, during, and after crash recovery for two client running concurrently. Client 1 requests exclusively the set of data that is on the server that will be killed. Client 2 requests the rest of the data.

Figure 3.11a shows the average CPU usage of the cluster. As we already explained in section II, RAMCloud monopolizes one core for its polling mechanism, which results in our case in a 25% CPU usage even when the system is idle. When the crash occurs the CPU usage jumps to 92%, then gradually decreases. This is mainly due to loading data from disks and replaying it at the same time. Figure 3.11b shows the overall average power consumption per node for the same scenario. The jump in resource usage translates in a power consumption of 119W. Since the power measured every second is an average, the increase in power consumption is not as sudden as the CPU usage.

The overhead of crash-recovery on normal operations. Figure 3.12 shows the average latency of two clients requesting data in parallel in the same configuration as the previous scenario. However, we chose manually a server to kill and make sure one of the clients always requests the subset of data held by that server. It is interesting first to notice that after the crash happens, the client which requests lost data is blocked for the whole duration of crash recovery, i.e., 40 seconds. Second, it is important to point out the jump in latency for the client which requests live data, i.e., from 15 us to 35 us just after the crash recovery starts. The average increase is between 1.4x and 2.4x in latency during crash recovery. This can be explained if we consider that in Figure 3.11a crash recovery alone causes up to 92% CPU usage. Consequently, normal operation are impacted during crash recovery.

Conclusion. As expected, crash recovery induces CPU (in addition to network and disk) overhead and up to 8% additional power consumption per node. We find that during crash recovery: (1) lost data is unavailable, causing latencies equal to the recovery time (e.g., 40 seconds for replication factor 4); (2) The performance of normal operations can have up to 2.4x additional latency in average due to the overhead of crash recovery.

Replication impact on crash-recovery. Random replication in RAMCloud intends to scatter data across the cluster to have as much resources as possible involved in recovery.

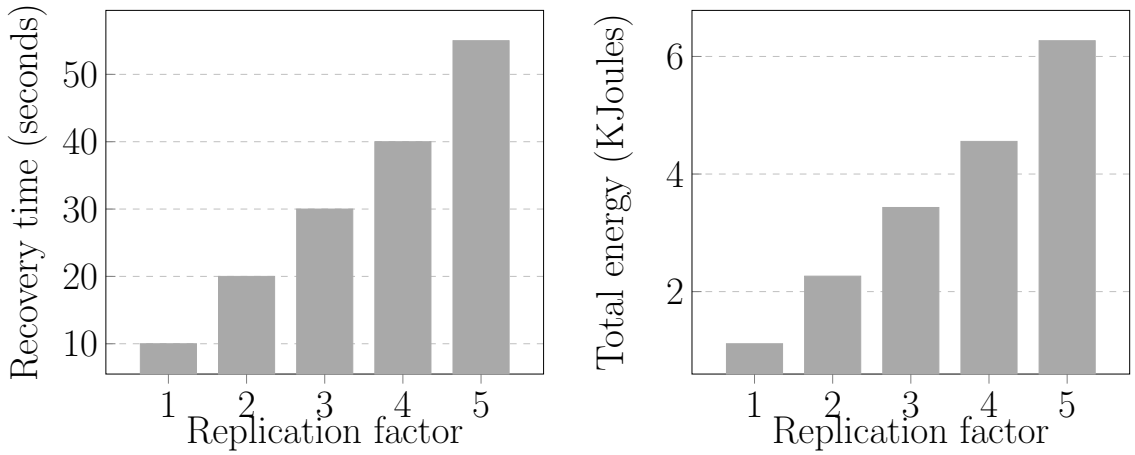


Figure 3.13 – (a) Recovery time as a function of the replication factor. (b) Average total energy consumption of a single node during crash recovery as a factor of the replication factor. The size of data to recover corresponds to 1.085 GB in both cases.

We have setup an experiment with 9 nodes and inserted 10M records which corresponds to 9.765 GB of data. Since we are inserting data uniformly in the RAMCloud cluster, each server has up to 1.085 GB of data. Same as previous experiment, we choose a random node after 1 min, then we kill RAMCloud process on that node.

Figure 3.13a shows the recovery time taken to reconstruct data of the crashed node. Surprisingly, increasing the replication factor increases the recovery time. With a replication factor of 1, only 10 seconds are needed to reconstruct data, this number almost grows linearly with the replication factor up to replication factor 5, where the time needed to reconstruct the same amount of data takes 55 seconds.

In order to shed the light on these results it is important to recall in more details how crash recovery works in RAMCloud. When a server is suspected to be crashed, the coordinator will check whether that server truly crashed. If it happens to be the case, the coordinator will schedule a recovery, after checking that the data held by that server is available on backups. Recovery happens on servers selected a-priori. After these steps, the recovery starts, first by reading lost data from backups disks, then replaying that data as in normal insert operations, i.e., inserting in DRAM, replicating it to backup replicas, waiting for acknowledgement and so on.

Impact of disk contention. While investigating the factors impacting the crash recovery performance we find that disk contention can have an impact. For instance Figure 3.14 corresponds to the total aggregated disk activity of the servers during crash recovery. We can see that right after the crash, there is a small increase in the read activity corresponding to reading backup data. Shortly after, there is a huge peak in write activity corresponding to the re-replication of the lost data. Read and write activities continue in parallel until the end of crash recovery.

We think that at small scale this issue is exacerbated. Since the probability of disk-interference between the backup performing a recovery, i.e., reading, and a server replaying

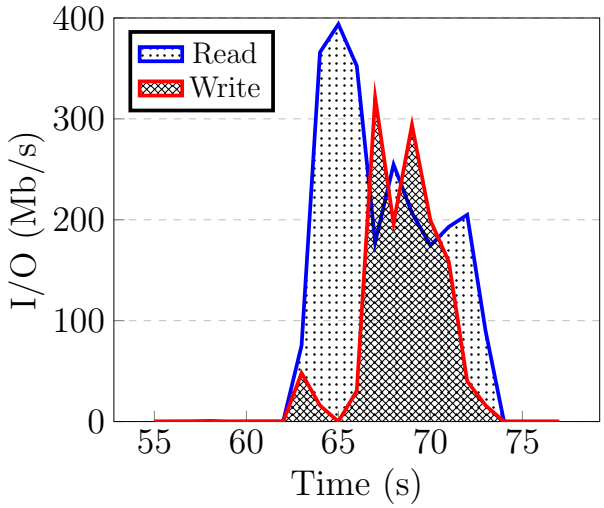


Figure 3.14 – Total aggregated disk activity (read and write) of 9 nodes during crash recovery. The size of data to recover corresponds to 1.085 GB. The dark part is the overlap between reads and writes.

data, i.e., writing, is high.

Conclusion. Counterintuitively, increasing the replication factor badly impacts the performance and the energy consumption of crash-recovery in RAMCloud. The replication factor should be carefully tuned according to the cluster size in order to avoid additional penalty due to replication.

In Figure 3.13b we report the total energy consumed by a single node during crash recovery within the precedent experiment. As expected, the energy consumed grows when increasing the replication factor. The energy increases almost linearly with the replication factor. It is noteworthy that the average power consumption of a node is comprised between 114 and 117 watts during recovery. Thus, the main factor leading to increased energy consumption is the additional time took to replay the lost data.

3.5 Conclusion

Recently, in-memory storage systems have become popular with the decrease of per-byte DRAM price. Researchers have focused mainly on improving the performance in-memory storage systems. In this chapter we study a well-known in-memory storage system. Through an extensive experimental evaluation, we focus on the trade-offs between performance and energy efficiency.

Firstly, we reveal that although RAMCloud scales linearly in throughput for read-only applications, it has a non-proportional power consumption. Mainly because it exhibits the same CPU usage under different levels of access.

Secondly, we show that prevalent Web workloads [59], i.e., read-heavy and update-heavy workloads, can impact significantly the performance and the energy consumption. We relate

it to the impact of concurrency, i.e., RAMCloud poorly handles its threads under highly-concurrent accesses.

Thirdly, we show that replication can be a major bottleneck for performance and energy. With update-heavy workloads, it can lead to 68% throughput degradation and 3.5x more energy consumption when increasing the replication factor from 1 to 4. The root cause of this issue is the contention at the server level between clients' requests and replication requests. Moreover, the replication scheme, which implies to wait for acknowledgements from all backups exacerbates this problem.

Finally, we quantify the overhead of a crash-recovery, which can end up in 90% CPU usage and 8% more power consumption on its own. We study the impact of replication on this mechanism and find that, surprisingly, it has a negative effect on it, i.e., increasing the replication factor increases recovery time and the energy consumption.

In the next chapter, we introduce an approach to mitigate some of these shortcoming. Specifically, we show how state-of-the-art networking technologies can be leveraged to reduce replication overheads in in-memory storage systems.

Chapter 4

Towards Efficient Replication for In-memory Storage: Tailwind

Contents

4.1 The Cost of Replication

In the last chapter we have shown that current durability and availability techniques can significantly impact the performance and energy efficiency of in-memory storage systems. In-memory storage systems, like many other systems, must replicate data in order to survive failures. As the core frequency scaling and multi-core architecture scaling are both slowing down, it becomes critical to reduce replication overheads to keep-up with shifting application workloads in key-value stores [47]. We show that replication can consume up to 80% of the CPU cycles for write-intensive workloads (§ 4.3.6.1), in strongly-consistent in-memory key-value stores.

Techniques like remote-direct memory access (RDMA) are promising to improve overall CPU efficiency of replication and keep predictable tail latencies. However, even existing RDMA-based approaches do not effectively use RDMA, as they require active backups. This guarantees the atomicity of RDMA transfers, since only fully received messages are applied by the receiver [21, 40]. However, this approach defeats RDMA efficiency goals, since it forces receivers to use their CPU to handle incoming RDMA messages and it incurs additional memory copies.

In this chapter we present *Tailwind*, a zero-copy primary-backup RDMA-based log replication protocol that completely bypasses CPUs on all target backup servers.

4.2 The Promise of RDMA and Challenges

Replication and redundancy are fundamental to fault tolerance, but at the same time they are costly. Primary-backup replication (PBR) is popular in fault-tolerant storage systems like file systems and key-value stores, since it tolerates f stop-failures with $f + 1$ replicas. Note that, we refer to a primary replica server as *primary*, and secondary replica server as *secondary* or *backup*. In some systems, backup servers don't process user-facing requests, but in many systems each node acts as both a primary for some data items and as a backup for other data items. In some systems this is implicit: for example, a key-value store may store its state on HDFS [76], and a single physical machine might run both a key-value store frontend and an HDFS chunkserver.

Replication is expensive for three reasons. First, it is inherently redundant and, hence, brings overhead: the act of replication itself requires moving data over the network. Second, replication in strongly consistent systems is usually synchronous, so a primary must stall while holding resources while waiting for acknowledgements from backups (often spinning a CPU core in low-latency stores). Third, in systems, where servers (either explicitly or implicitly) serve both client-facing requests and replication operations, those operations contend.

Figure 4.1 shows this in more detail. Low-latency, high-throughput stores use kernel-bypass to directly poll NIC control rings (with a dispatch core) to avoid kernel code paths and interrupt latency and throughput costs. Even so, a CPU on a primary node processing an update operation must receive the request, hand the request off to a core (worker core) to be processed, send remote messages, and then wait for multiple nodes acting as backup to process these requests. Batching can improve the number of backup request messages each server must receive, but at the cost of increased latency. Inherently, though, replication can

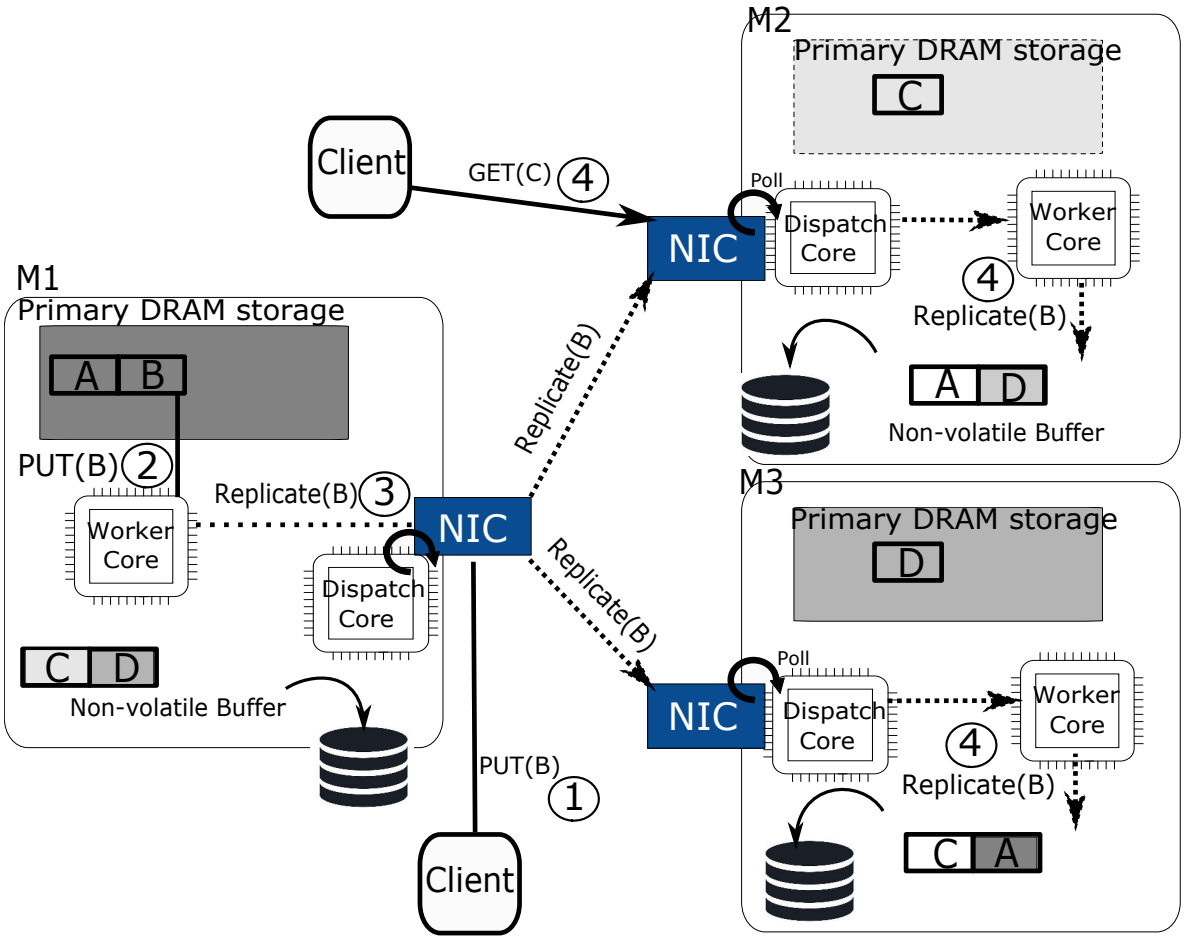


Figure 4.1 – Flow of primary-backup replication

double, triple, or quadruple the number of messages and the amount of data generated by client-issued write requests. It also causes expensive stalls at the primary while it waits for responses. In these systems, responses take a few microseconds which is too short a time for the primary to context switch to another thread, yet its long enough that the worker core spends a large fraction of its time waiting.

4.2.1 Background on RDMA

Remote Direct Memory Access (RDMA) allows one computer to directly access the memory of a remote computer without involving the operating system at any host. This enables zero-copy transfers, reducing latency and CPU overhead. In this work, we focus on two types of RDMA-providing interconnects: InfiniBand and RoCE (RDMA over Converged Ethernet). However, we believe that our design is applicable to other RDMA providers such as iWARP, Quadrics, and Myrinet.

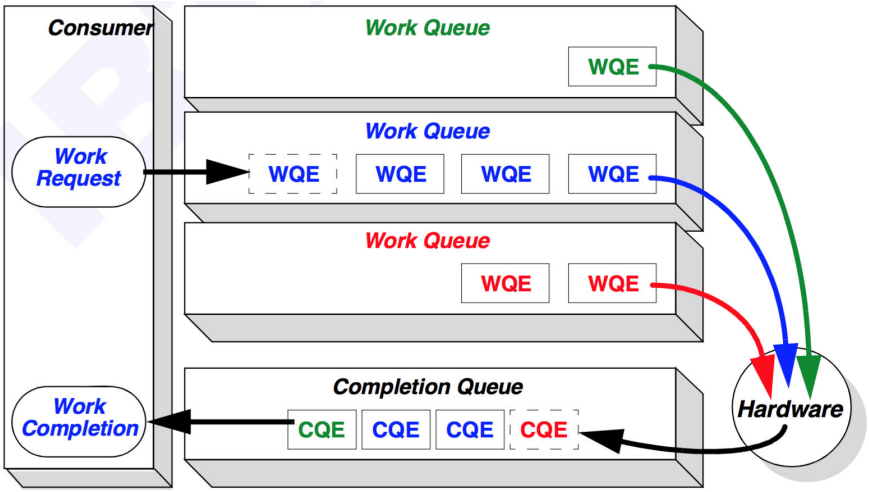


Figure 4.2 – IB verbs queuing model [31]

4.2.1.1 Verbs and Queue Pairs.

RDMA was originally supported by the Infiniband high-speed network fabric, commonly used in high-performance computing. A user program can directly access the host channel adapter (HCA) via a *verb interface*. There are several types of verbs:

- **Infiniband.** Infiniband verbs are the most widely used type of verbs to create RDMA-based applications [31].
- **iWARP.** Internet Wide Area RDMA Protocol (iWARP) enables application to place data, remotely, to the target without involving its CPU.
- **RoCE.** RDMA over Converged Ethernet (RoCE) provides an implementation of RDMA just on top of Ethernet.

4.2.1.2 IB verbs

The main operations that the verbs API offer are SEND, RECEIVE (RECV) RDMA WRITE (WRITE) and RDMA READ (READ). READ and WRITE operations have memory semantics: they specify a remote memory address to read from or write to. They don't invlove the remote CPU, i.e., the receive is idle during the whole process. These type of verbs are often called *one-sided* RDMA. On the other hand SEND and RECV do involve the remote CPU and are called *two-sided* RDMA. These operations are referred to as having channel semantics. For instance, the SEND operation pushes data to the remote side, and the destination chooses where to place the data.

4.2.1.3 Queue Pairs

By nature, IB verbs operations follow an asynchronous model. For instance, a sender might post many request to be executed by the hardware. The queue used to hold these services

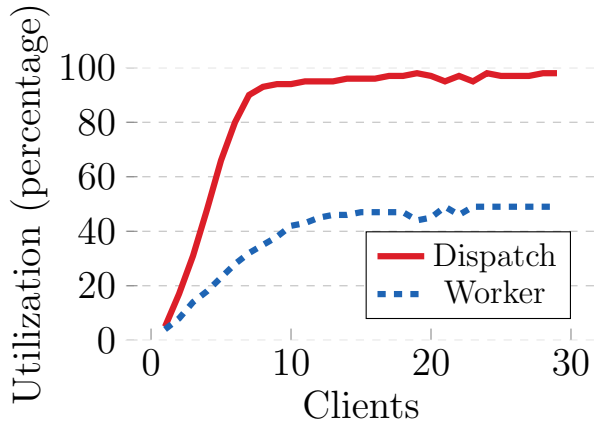


Figure 4.3 – Example of CPU utilization of a single RAMCloud server when running the YCSB benchmark. Requests consist of 95/5 GET/PUT ratio.

requests is referred to as a work queue [31]. Work queues are typically created in pairs, i.e. Queue Pair (QP), one for send operations and one for receive operations. In general, from a sender node perspective, the send work queue holds instructions that cause data to be transferred between the sender memory and another node memory. The receive work queue holds instructions about where to place data that is received from another node. More specifically, a sender who wishes to initiate a SEND submits a work request (WR), which causes an instruction called a Work Queue Element (WQE) to be placed on the appropriate work queue, as depicted in Figure 4.2. When the channel adapter completes a WQE, a Completion Queue Element (CQE) is placed on a completion queue.

4.2.2 RDMA Opportunities

One-sided RDMA operations are attractive for replication; replication inherently requires expensive, redundant data movement. Backups are (mostly) passive; they often act as dumb storage, so they may not need CPU involvement. Figure 4.3 shows that RAMCloud, an in-memory low-latency kernel-bypass-based key-value store, is often bottlenecked on CPU (see § 4.3.4 for experimental settings). For read-heavy workloads, the cost of polling network and dispatching requests to idle worker cores dominates. Only 8 clients are enough to saturate a single server dispatch core. Because of that, worker cores cannot be fully utilized. One-sided operations for replicating PUT operations would reduce the number of requests each server handles in RAMCloud, which would indirectly but significantly improve read throughput. For workloads with a significant fraction of writes or where a large amount of data is transferred, write throughput can be improved, since remote CPUs needn't copy data between NIC receive buffers and I/O or non-volatile storage buffers.

4.2.3 Challenges

The key challenge in using one-sided RDMA operations is that they have simple semantics which offer little control on the remote side. This is by design; the remote NIC executes RDMA operations directly, so they lack the generality that a conventional CPU-based RPC

handlers would have. A host can issue a remote *read* of a single, sequential region of the remote processes virtual address space (the region to read must be *registered* first, but a process could register its whole virtual address space). Or, a host can issue a remote *write* of a single, sequential region of the remote processes virtual address space (again, the region must be registered with the NIC). NICs support a few more complex operations (compare-and-swap, atomic add), but these operations are currently much slower than issuing an equivalent two-sided operation that is serviced by the remote CPU [82, 39]. These simple, restricted semantics make RDMA operations efficient, but they also make them hard to use safely and correctly. Some existing systems use one-sided RDMA operations for replication (and some also even use them for normal case operations [21, 22]).

However, no existing primary-backup replication scheme reaps the full benefits of one-sided operations. In existing approaches, source nodes send replication operations using RDMA writes to push data into ring buffers. CPUs at backups poll for these operations and apply them to replicas. In practice, this is is effectively emulating two-sided operations [21]. RDMA reads don't work well for replication, because they would require backup CPUs to schedule operations and "pull" data, and primaries wouldn't immediately know when data was safely replicated.

Two key, interrelated issues make it hard to use RDMA writes for replication that fully avoids the remote CPUs at backups. First, a primary can crash when replicating data to a backup. Because RDMA writes (inherently) don't buffer all of the data to be written to remote memory, it is possible that an RDMA write could be partially applied when the primary crashes. If a primary crashes while updating state on the backup, the backup's replica wouldn't be in the "before" or "after" state, which could result in a corrupted replica. Worse, since the primary was likely mutating all replicas concurrently, it is possible for all replicas to be corrupted. Interestingly, backup crashes during RDMA writes don't create new challenges for replication, since protocols must deal with that case with conventional two-sided operations too. Well-known techniques like log-structured backups [56, 72, 64] or shadow paging [94] can be used to prevent update-in-place and loss of atomicity. Traditional log implementations enforce a total ordering of log entries [34]. In database systems, for instance, the order is used to recreate a consistent state during recovery.

Unfortunately, a second key issue with RDMA operations makes this hard: each operation can only affect a single, contiguous region of remote memory. To be efficient, one-sided writes must replicate data in its final, stable form, otherwise backup CPU must be involved, which defeats the purpose. For stable storage, this generally requires some metadata. For example, when a backup uses data found in memory or storage it must know which portions of memory contain valid objects, and it must be able to verify that the objects and the markers that delineate them haven't been corrupted. As a result, backups need some metadata about the objects that they host in addition to the data items themselves. However, RDMA writes make this hard. Metadata must inherently be intermixed with data objects, since RDMA writes are contiguous. Otherwise, multiple round trips would be needed, again defeating the efficiency gains.

Next we present the design of Tailwind. We describe how it solves the above-cited challenges.

4.3 Tailwind

Tailwind is a strongly-consistent RDMA-based replication protocol. It was designed to meet four requirements:

Zero-copy, Zero-CPU on Backups for Data Path. In order to relieve backups CPUs from processing replication requests, Tailwind relies on one-sided RDMA writes for all data movement. In addition, it is zero-copy at primary and secondary replicas; the sender uses kernel-bypass and scatter/gather DMA for data transfer; on the backup side, data is directly placed to its final storage location via DMA transfer without CPU involvement.

Strong Consistency. For every object write Tailwind synchronously waits for its replication on all backups before notifying the client. Although RDMA writes are one-sided, reliable-connected QPs generate a work completion to notify the sender once a message has been correctly sent and acknowledged by the receiver NIC (i.e. written to remote memory) [31]. One-sided operation raise many issues, Tailwind is designed to cover *all* corner cases that may challenge correctness (§ 4.3.3.3).

Overhead-free Fault-Tolerance. Backups are unaware of replication as it happens, which can be unsafe in case of failures. To address this, Tailwind appends a piece of metadata in the log after every object update. Backups use this metadata to check integrity and locate valid objects during recovery. Although a few backups have to do little extra work during crash recovery, that work has no impact on recovery performance (§ 4.3.8).

Preserves Client-facing RPC Interface. Tailwind has no requirement on the client side; all logic is implemented between primaries and backups. Clients observe the same consistency guarantees. However, for write operations, Tailwind highly improves end-to-end latency and throughput from the client perspective.

4.3.1 The Metadata Challenge

Metadata is crucial for backups to be able to use replicated data. For instance, a backup needs to know which portions of the log contain valid data. In RPC-based systems, metadata is usually piggybacked within a replication request [61, 39]. However, it is challenging to update both data and metadata with a single RDMA write since it can only affect a contiguous memory region. In this case, updating both data and metadata would require sending two messages which would nullify one-sided RDMA benefits. Moreover, this is risky: in case of failures a primary may start updating the metadata and fail before finishing, thereby invalidating all replicated objects.

For log-structured data, backups need two pieces of information: (1) the *offset* through which data in the buffer is valid. This is needed to guarantee the atomicity of each update. An outdated offset may lead the backup to use old and inconsistent data during crash recovery. (2) A *checksum* used to check the integrity of the length fields of each log record during recovery. Checksums are critical for ensuring log entry headers are not corrupted while in buffers or on storage. These checksums ensure iterating over the buffer is safe; that is, a

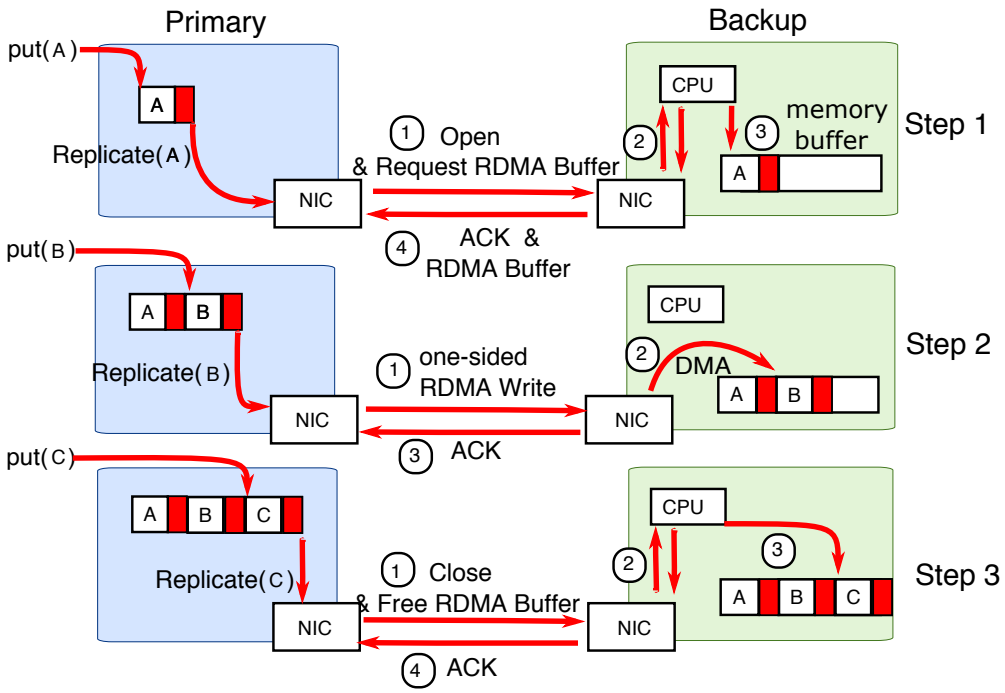


Figure 4.4 – The three replication steps in Tailwind. During the first (open) and third (close) steps, the communication is done through RPCs. While the second step involves one-sided writes only.

corrupted length field does not "point" into the middle of another object, out of buffer, or indicate an early end to the buffer.

The protocol assumes that each object has a *header* next to it [61, 44, 21]. Implementation-agnostic information in headers should include: (1) the size of the object next to it to allow log traversal; (2) an integrity check that ensures the integrity of the contents of the log entry.

Tailwind checksums are 32-bit CRCs computed over log entry headers. The last checksum in the buffer covers all previous headers in the buffer. For maximum protection, checksums are end-to-end: they should cover the data while it is in transit over the network and while it occupies storage.

To be able to perform atomic updates with one-sided RDMAs in backups, the last checksum and the current offset in the buffer must be present and consistent in the backup after *every* update. A simple solution is to append the checksum and the offset before or after every object update. A single RDMA write would suffice then for both data and metadata. The checksum *must* necessarily be sent to the backup. Interestingly, this is not the case for the offset. The nature of log-structured data and the properties of one-sided RDMA make it possible, with careful design, for the backup to compute this value at recovery time without hurting consistency. This is possible because RDMA writes are performed (at the receiver side) in an increasing address order [31]. In addition, reliable-connected QPs ensure that updates are applied in the order they were sent.

Based on these observations, Tailwind appends a checksum in the log after every object update; at any point of time a checksum guarantees, with high probability, the integrity of all previous headers preceding it in the buffer. During failure-free time, a backup is ensured

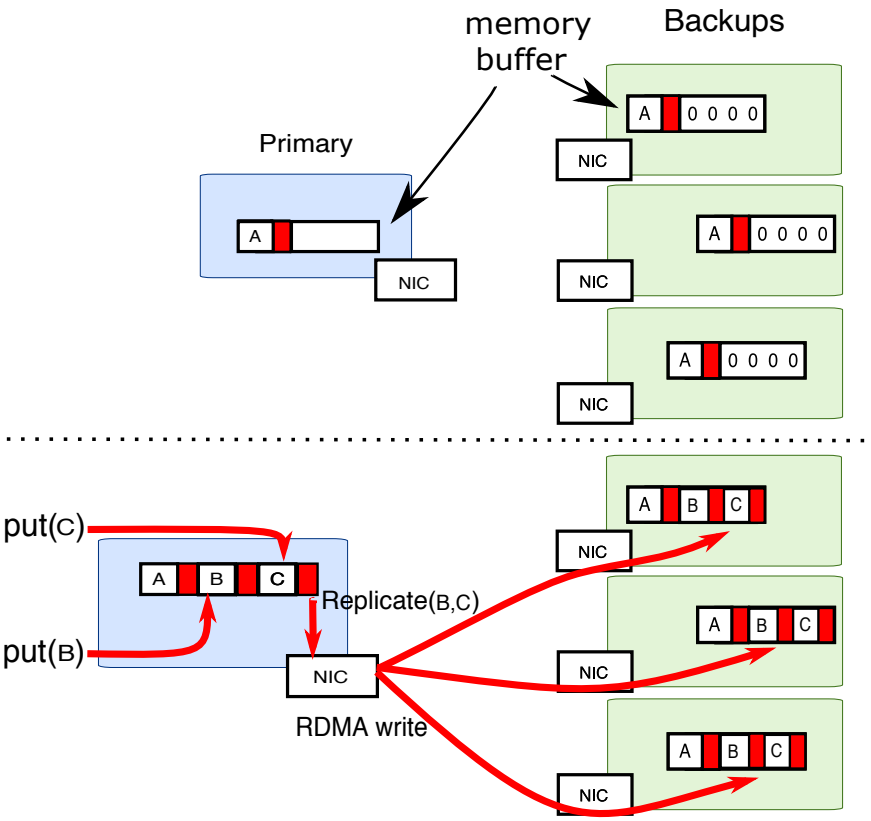


Figure 4.5 – Memory buffers contents on a primary and secondary replicas are always identical. Buffers are pre-zeroed on backups. Primaries send RDMA write to remote NICs that perform DMA in increasing ad- dress order [31]. Plain rectangles after each object represent a checksum of all preceding object headers in the buffer.

to always have the latest checksum at the end of the log. On the other hand, backups have to compute the offset themselves during crash recovery.

4.3.2 Non-volatile Buffers

In Tailwind, at start up, each backup machine allocates a pool of in-memory I/O buffers (8 MB each, by default) and registers them with the NIC. To guarantee safety, each backup limits the number of buffers outstanding unflushed buffers it allows. This limit is a function of its local, durable storage speed. A backup denies opening a new replication buffer for a primary if it would exceed the amount of data it could flush safely to storage on backup power. Buffers are pre-zeroed as show in Figure 4.5. Servers require power supplies that allow buffers to be flushed to stable storage in the case of a power failure [61, 21, 22]. This avoids the cost of a synchronous disk write on the fast path of PUT operations.

Initiatives such as the OpenCompute Project propose standards where racks are backed by batteries backup, that could provide a minimum of 45 seconds of power supply [86] at full load, including network switches. Battery-backed DIMMs could have been another option, but they require more careful attention. Because we use RDMA, batteries need to back the CPU, the PCIe controller, and the memory itself. Moreover, there exists no clear way to flush

data that could still be residing in NIC cache or in PCIe controller, which would lead to firmware modifications and to a non-portable solution.

4.3.3 Replication Protocol

4.3.3.1 Write Path

To be able to perform replication through RDMA, a primary has to has to reserve an RDMA-registered memory buffer from a secondary replica. The first step in Figure 4.4 depicts this operation: a primary sends an `open` RPC to a backup (1). Tailwind does not enforce any replica placement policy, instead it leaves backup selection up to the storage system. Once the `open` is processed (2) + (3), the backup sends an acknowledgement to the primary and piggybacks necessary information to perform RDMA writes (4) (i.e. `remote_key` and `remote_address` [31]). The open call fails if there are no available buffers. The primary has then to retry elsewhere.

At the second step in Figure 4.4, the primary is able to perform all subsequent replication requests with RDMA writes (1). The backup NIC directly put objects into memory buffers via DMA transfer (2) without involving the CPU. The primary gets work completion notifications from its corresponding QP (3).

The primary keeps track of the last written offset in the backup buffer. When the next object would exceed the buffer capacity, the primary proceeds to the third step in Figure 4.4. The last replication request is called `close` and is performed through an RPC (1). The `close` notifies the backup (2) + (3) that the buffer is full and thus can be flushed to durable storage. This eventually allows Tailwind to reclaim buffers and make them available to other primaries. Buffers are zeroed again when flushed.

We use RPCs for `open` and `close` operations because it simplifies the design of the protocol without hurting latency. As an example of complication, a primary may choose a secondary that has no buffers left. This can be challenging to handle with RDMA. Moreover, these operations are not frequent. If we consider 8 MB buffers and objects of 100 B, which corresponds to real workloads object size [59], `open` and `close` RPCs would account for 2.38×10^{-5} of the replication requests. Larger buffers imply less RPCs but longer times to flush backup data to secondary storage.

Thanks to all these steps, Tailwind can effectively replicate data using one-sided RDMA. However, without taking care of failure scenarios the protocol would not be correct. Next, we define essential notions Tailwind relies on for its correctness.

4.3.3.2 Primary Memory Storage

The primary DRAM log-based storage is logically sliced into equal *segments* (Figure 4.6). For every `open` and `close` RPC the primary sends metadata information about its current state: *logID*, latest checksum, *segmentID*, and current offset in the last segment. In case of failures, this information helps the backup in finding backup-data it stores for the crashed server.

At any given time, a primary can only replicate a single segment to its corresponding backups. This means a backup has to do very little work during recovery; if a primary replicates to r replicas then only r segments would be open in case the primary fails.

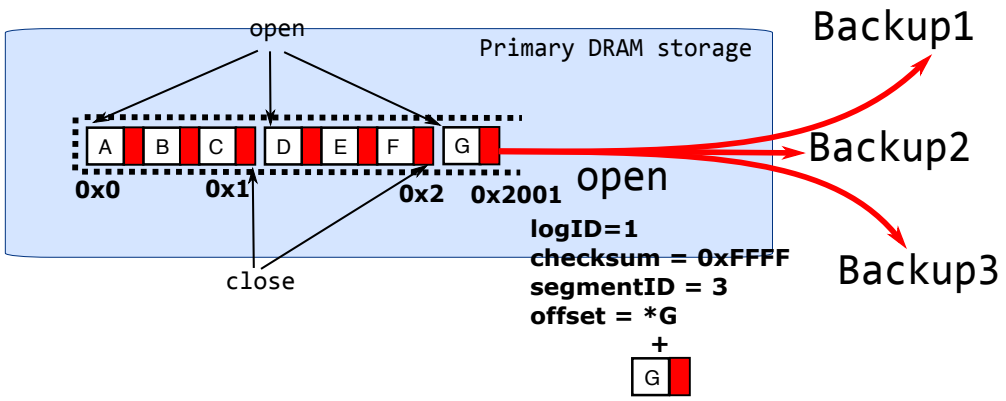


Figure 4.6 – Primary DRAM storage consists of a monotonically growing log. It is logically split into fixed-size segments.

4.3.3.3 Failure Scenarios

When a primary or secondary replica fails, the protocol must recover from the failure and rebuild lost data. Primary and secondary replicas failure scenarios require different actions to recover.

4.3.3.4 Primary-replica Failure

Primary replica crashes are one of the major concerns in the design. In case of such failures, Tailwind has to: (1) locate backup data (of crashed primary) on secondary replicas; (2) rebuild up-to-date metadata information on secondary replicas; (3) ensure backup data integrity and consistency; (4) start recovery.

Locating Secondary Replicas. After detecting a primary replica crash, Tailwind sends a query to all secondary replicas to identify the ones storing data belonging to the crashed primary. Since every primary has a unique logID it is easy for backups to identify which buffers belong to that logID.

Building Up-to-date Metadata. Backup buffers can either be in `open` or `close` states. Buffers that are `closed` do not pose any issue, they already have up-to-date metadata. If they are in disk or SSD they will be loaded to memory to get ready for recovery. However, for `open` buffers, the backup has to compute the offset and find the last checksum. Secondary replicas have to **scan** their `open` buffers to update their respective checksum and offset. To do so, they iterate over all entries as depicted in Algorithm 1.

Basically, the algorithm takes an `open` buffer and tries to iterate over its entries. It moves forward thanks to the size of the entry which should be in the header. For every entry the backup computes a checksum over the header. When reaching a checksum in the buffer it is compared with the most recently computed checksum: the algorithm stops in case of checksum mismatch. There are three stop conditions: (1) integrity check failure; (2) invalid object found; (3) end of buffer reached.

```
   input  : Pointer to a memory buffer rdmaBuf
   output: Size of durably replicated data offset
 1 currPosition = rdmaBuf ;
 2 offset = currPosition ;
 3 while currPosition < MAX_BUFFER_SIZE do
       /* Create a header in the current position                    */
 4     header = (Header)currPosition ;
 5     currPosition += sizeof(header) ;
       /* Not Corrupted or incomplete header                         */
 6     if header→type != INVALID then
 7         if header→type == checksumType then
 8             checksum = (Checksum)currPosition ;
 9             if checksum != currChecksum then
                   /* Primaries never append a zero checksum, check if
                      it is 1.                                        */
10                 if currChecksum == 0 and checksum == 1 then
11                     offset = currPosition +
12                     sizeOf(checksum) ;
13                 else
14                     return offset ;
15             else
                   /* Move the offset at the end of current checksum  */
16                 offset = currPosition + sizeOf(checksum);
17         else
18             currChecksum = crc32(currChecksum, header);
19     else
20         return offset;
       /* Move forward to next entry                                 */
21     currPosition += header→objectSize;
   /* We should only reach this line if a primary crashed before
      sending close                                                   */
22 return offset;
```

Algorithm 1: Updating RDMA buffer metadata.

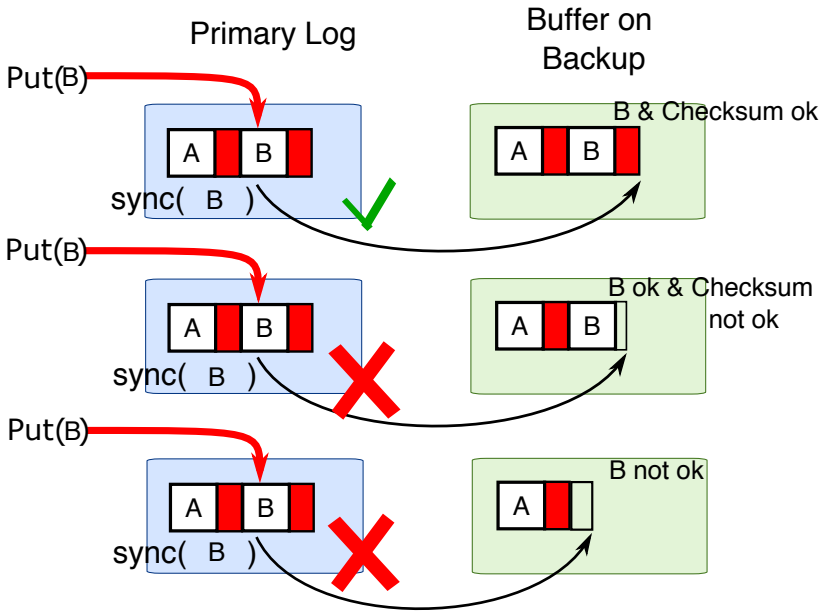


Figure 4.7 – From top to bottom are three scenarios that can happen when a primary replica crashes while writing an object (B in this case) then synchronizing with backups. In the first scenario the primary replica fully writes the message to the backup leaving the backup in a correct state. B can be recovered in this case. In the second scenario, the object B is written but the checksum is partially written. Therefore, B is discarded. Similarly for the third scenario where B is partially written.

A combination of three factors guarantee the correctness of the algorithm: (1) **the last entry is always a checksum**; Tailwind implicitly uses this condition as an end-of-transmission marker. (2) **Checksums are not allowed to be zero**; the primary replica always verifies the checksum before appending it. If it is zero it sets it to 1 and then appends it to the log. Otherwise, an incomplete object could be interpreted as valid zero checksum. (3) **Buffers are pre-zeroed**; combined with condition (2), a backup has a means to correctly detect the last valid offset in a buffer by using Algorithm 1.

4.3.3.5 Corrupted and Incomplete Objects

Figure 4.7 shows the three states of a backup RDMA buffer in case a primary replica failure. The first scenario shows a successful transmission of an object B and the checksum ahead of it. If the primary crashes, the backup is able to safely recover all data (i.e. A and B).

In the second scenario the backup received B, but the checksum was not completely received. In this case the integrity check will fail. Object A will be recovered and B will be ignored. This is safe, since the client's PUT of B could not have been acknowledged.

The third scenario is similar: B was not completely transmitted to the backup. However, there creates two possibilities. If B's header was fully transmitted, then the algorithm will look past the entry and find a 0-byte at the end of the log entry. This way it can tell that the RDMA operation didn't complete in full, so it will discard the entry and treat the prefix of the buffer up through A as valid. If the checksum is partially written, it will still be discarded,

since it will necessarily end in a 0-byte: something that is disallowed for all valid checksums that the primary creates. If B's header was only partially written, some of the bytes of the length field may be left zero. Imagine that o is the offset of the start of B. If the primary intended B to have length l and l' is the length actually written into the backup buffer. It is necessarily the case that $l' < l$, since lengths are unsigned and l' is a subset of the bits in l. As a result, $o + l'$ falls within the range where the original object data should have been replicated in the buffer. Furthermore, the data there consists entirely of zeroes, since an unsuccessful RDMA write halts replication to the buffer, and replication already halted before $o +$ `sizeof(Header)`. As a result, this case is handled identically to the incomplete checksum case, leaving the (unacknowledged) B off of the valid prefix of the buffer.

A key property maintained by Tailwind is that torn writes never depend on checksum checks for correctness. They can also be detected by careful construction of the log entries headers and checksums and the ordering guarantees that RDMA NICs provide.

Bit-flips The checksums, both covering the log entry headers and the individual objects themselves ensure that recovery is robust against bit-flips. The checksums ensure with high probability that bit-flip anywhere in any replica will be detected. In `closed` segments, whenever data corruption is detected, Tailwind declares the replica corrupted. The higher-level system will still successfully recover a failed primary, but it must rely on replicas from other backups to do so. In `open` segments data corruption is treated as partially transmitted buffers; as soon as Tailwind immediately stops iterating over the buffer and returns the last valid offset.

4.3.3.6 Secondary-replica Failure

When a server crashes the replicas it contained become unavailable. Tailwind must re-create new replicas on other backups in order to keep the same level of durability. Luckily, secondary-replica crashes are dealt with naturally in storage systems and do not suffer from one-sided RDMA complications. Tailwind takes several steps to allocate a new replica: (1) it suspends all operations on the corresponding primary replica; (2) it **atomically** creates a new secondary replica; (3) it resumes normal operations on the primary replica. Step (1) ensures that data will always have the same level of durability. Step (2) is crucial to avoid inconsistencies if a primary crashes while re-creating a secondary replica. In this case the newly created secondary replica would not have all data and cannot be used.

However, it can happen that a secondary replica stops and restarts after some time, which could lead to inconsistent states between secondary replicas. To cope with this, Tailwind keeps, at any point of time, a version number for replicas. If a secondary replica crashes, Tailwind updates the version number on the primary and secondaries. Since secondaries need to be notified, Tailwind uses an RPC instead of an RDMA for this operation. Tailwind updates the version number right after the step (2) when re-creating a secondary replica. This ensures that the primary and backups are synchronized. Replication can start again from a consistent state. Note that this RPC is rare and only occurs after the crash of a backup.

4.3.3.7 Network Partitioning

It can happen that a primary is considered crashed by a subset of servers. Tailwind would start locating its backups, then rebuilding metadata on those backups. While metadata is re-

CPU	Xeon E5-2450 2.1 GHz 8 cores, 16 hw threads
RAM	16 GB 1600 MHz DDR3
NIC	Mellanox MX354A CX3 @ 56 Gbps
Switch	36 port Mellanox SX6036G
OS	Ubuntu 15.04, Linux 3.19.0-16, MLX4 3.4.0, libibverbs 1.2.1

Table 4.1 – Experimental cluster configuration.

built, the primary could still perform one-sided RDMA to its backups, since they are always unaware of these type of operations. To remedy this, as soon as a primary or secondary failure is detected, all machines close their respective QPs with the crashed server. This allows Tailwind to ensure that servers that are alive but considered crashed by the environment do not interfere with work done during recovery.

4.3.4 Evaluation

We implemented Tailwind on RAMCloud a low-latency in-memory key-value store. Tailwind's design perfectly suits RAMCloud in many aspects:

Low latency. RAMCloud's main objective is to provide low-latency access to data. It relies on fast networking and kernel-bypass to provide a fast RPC layer. Tailwind can further improve RAMCloud (PUT) latency (§ 4.3.5.1) by employing one-sided RDMA without any additional complexity or resource usage.

Replication and Threading in RAMCloud. To achieve low latency, RAMCloud dedicates one core solely to poll network requests and dispatch them to worker cores (Figure 4.1). Worker cores execute all client and system tasks. They are never preempted to avoid context switches that may hurt latency. To provide strong consistency, RAMCloud always requests acknowledgements from all backups for every update. With the above threading-model, replication considerably slows down the overall performance of RAMCloud as we have showed in the last chapter. Hence Tailwind can greatly improve RAMCloud's CPU-efficiency and remove replication overheads.

Log-structured Memory. RAMCloud organizes its memory as an append-only log. Memory is sliced into smaller chunks called *segments* that also act as the unit of replication, i.e., for every segment a primary has to choose a backup. Such an abstraction makes it easy to replace RAMCloud's replication system with Tailwind. Tailwind checksums can be appended in the log-storage, with data, and replicated with minimal changes to the code. In addition, RAMCloud provides a log-cleaning mechanism which can efficiently clean old checksums and reclaim their storage space.

We compared Tailwind with RAMCloud replication protocol, focusing our analysis on three key questions:

Does Tailwind improve performance? Measurements show Tailwind reduces RAMCloud's median write latency by 2× and 99th percentile latency by 3× (Figure 4.9). Tailwind improves throughput by 70% for write-heavy workloads and by 27% for workloads that include just a small fraction of writes.

Why does Tailwind improve performance? Tailwind improves per-server throughput by eliminating backup request processing (Figure 4.11), which allows servers to focus effort on processing user-facing requests.

What is the Overhead of Tailwind? We show that Tailwind's performance improvement comes at no cost. Specifically, we measure and find no overhead during crash recovery compared to RAMCloud.

4.3.5 Experimental Setup

Experiments were done on a 35 server Dell r320 cluster (Table 4.1) on the CloudLab [70] testbed.

We used three YCSB [12] workloads to evaluate Tailwind: update-heavy (50% PUTs, 50% GETs), read-heavy (5% PUTs, 95% GETs), and update-only (100% PUTs). We intitially inserted 20 million objects of 100 B plus 30 B for the key. Afterwards, we ran up to 30 client machines. Clients generated requests according to a Zipfian distribution ($\theta = 0.99$). Objects were uniformly inserted in active servers. The replication factor was set to 3 and RDMA buffers size was set to 8 MB. Every data point in the experiments is averaged over 3 runs.

RAMCloud's RPC-based replication protocol served as a baseline for comparison. Note that, in the comparison with Tailwind, we refer to RAMCloud's replication protocol as RAMCloud for simplicity.

4.3.5.1 Performance Improvement

The primary goal of Tailwind is to accelerate basic operations' throughput and latency. To demonstrate how Tailwind improves performance we show Figure 4.8, i.e. throughput per server as we increase the number of clients. When client operations consist of 5% PUTs and 95% GETs, RAMCloud achieves 500 KOp/s per server while Tailwind reaches up to 635 KOp/s. Increasing the update load enables Tailwind to further improve the throughput. For instance with 50% PUTs Tailwind sustains 340 KOp/s against 200 KOp/s for RAMCloud, which is a 70% improvement. With update-only workload, improvement is not further increased: in this case Tailwind improves the throughput by 65%.

Tailwind can improve the number of read operations serviced by accelerating updates. CPU cycles saved allow servers (that are backups as well) to service more requests in general.

Figure 4.9 shows that update latency is also considerably improved by Tailwind. Under light load Tailwind reduces median and 99th percentile latency of an update from 16 µs to 11.8 µs and from 27 µs to 14 µs respectively. Under heavy load, i.e. 500 KOp/s Tailwind reduces median latency from 32 µs to 16 µs compared to RAMCloud. Under the same load tail latency is even further reduced from 78 µs to 28 µs.

Tailwind can effectively reduce end-to-end client latency. With reduced acknowledgements waiting time, and more CPU power to process requests faster, servers can sustain a very low latency even under heavy concurrent accesses.

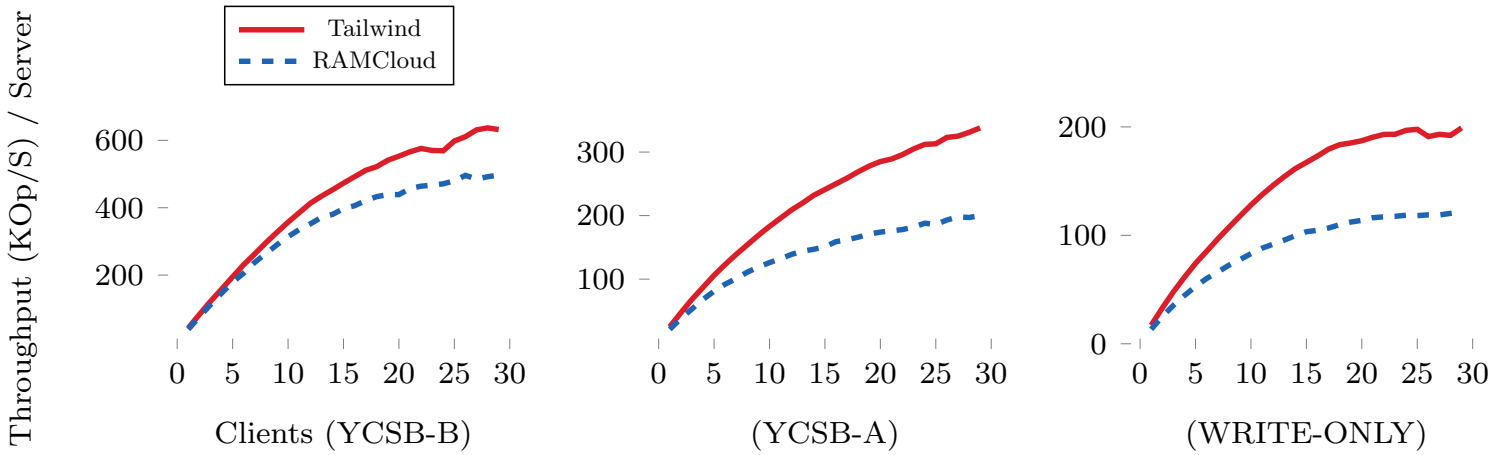


Figure 4.8 – Throughput per server in a 4 server cluster.

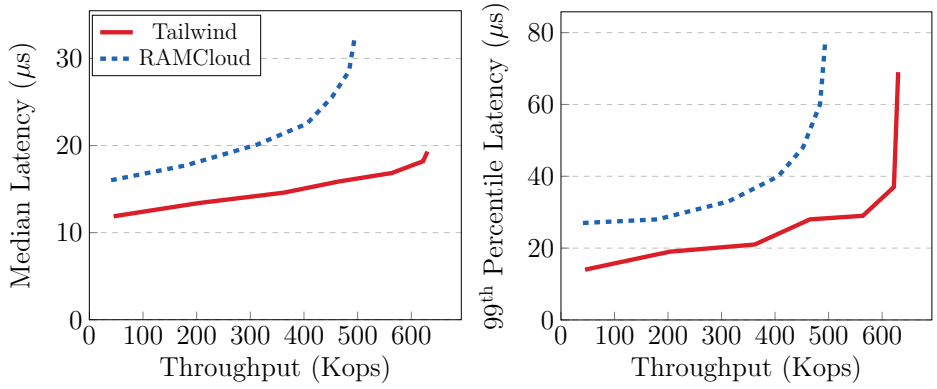


Figure 4.9 – (a) Median latency and (b) 99th percentile latency of PUT operations when varying the load.

4.3.6 Gains as Backup Load Varies

Since all servers in RAMCloud act as both backups and primaries, Tailwind accelerates normal-case request processing indirectly by eliminating the need for servers to actively process replication operations. Figure 4.10 shows the impact of this effect. In each trial load is directed at a subset of four RAMCloud storage nodes; "Active Primary Servers" indicates the number of storage nodes that process client requests. Nodes do not replicate data to themselves, so when only one primary is active it is receiving no backup operations. All of the active primary's backup operations are directed to the other three otherwise idle nodes. Note that, in this figure, throughput is per-active-primaries. So, as more primaries are added, the aggregate cluster throughput increases.

As client GET/PUT operations are directed to more nodes (more active primaries), each node slows down because it must serve a mix of client operations intermixed with an increasing number of incoming backup operations. Enough client load is offered (30 clients) so that storage nodes are the bottleneck at all points in the graphs. With four active primaries, every server node is saturated processing client requests and backup operations for all client-issued writes.

Even when only 5% of client-issued operations are writes (Figure 4.10a), Tailwind increasingly improves performance as backup load on nodes increases. When a primary doesn't perform backup operations Tailwind improves throughput 3%, but that increases to 27% when the primary services its fair share of backup operations. The situation is similar when client operations are a 50/50 mix of reads and writes (Figure 4.10b) and when clients only issue writes (Figure 4.10c).

As expected, Tailwind enables increasing gains over RAMCloud with increasing load, since RDMA eliminates three RPCs that each server must handle for each client-issued write, which, in turn, eliminates worker core stalls on the node handling the write.

In short, the ability of Tailwind to eliminate replication work on backups translates into more availability for normal request processing, and, hence, better GET/PUT performance.

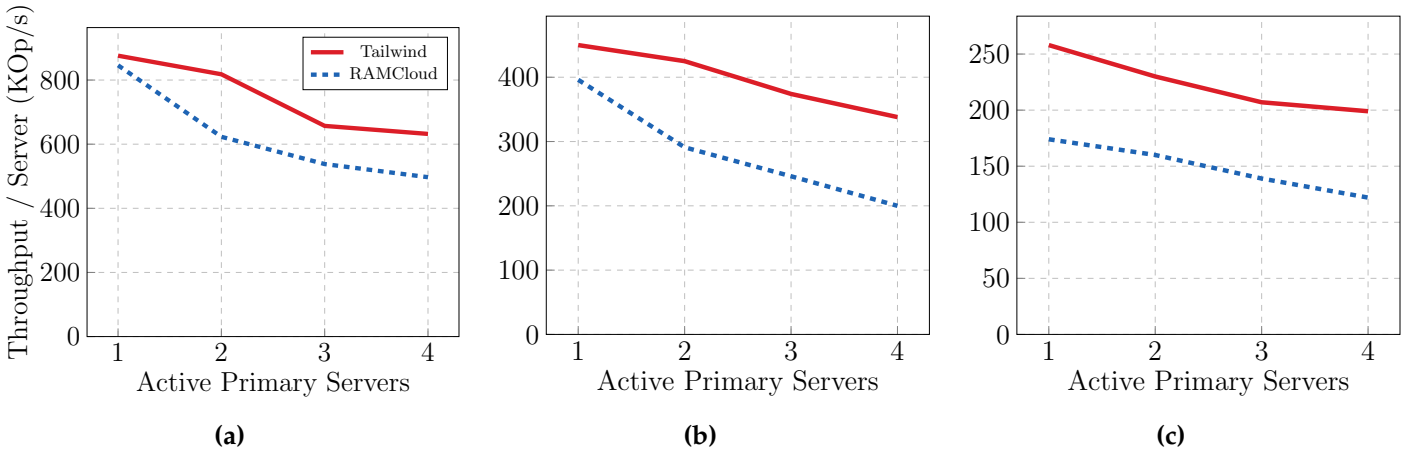


Figure 4.10 – Throughput per active primary servers when running (a) YCSB-B (b) YCSB-A (c) WRITE-ONLY with 30 clients.

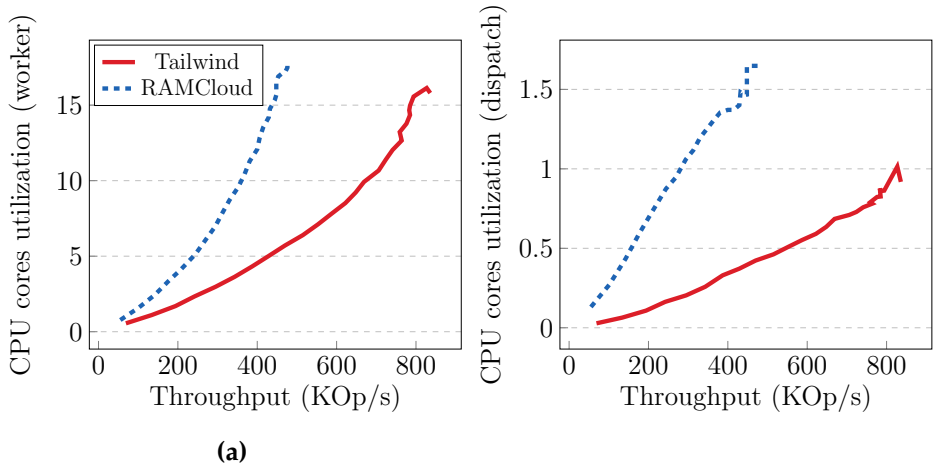


Figure 4.11 – Total (a) worker and (b) dispatch CPU core utilization on a 4-server cluster. Clients use the WRITE-ONLY workload.

4.3.6.1 Resource Utilization

The improvements above have shown how Tailwind can improve RAMCloud's baseline replication normal-case. The main reason is that operations contend with backup operations for worker cores to process them. Figure 4.11a illustrates this: we vary the offered load (updates-only) to a 4-server cluster and report aggregated active worker cores. For example, to service 450 KOp/s, Tailwind uses 5.7 worker cores while RAMCloud requires 17.6 active cores, that is 3× more resources. For the same scenario, we also show Figure 4.11b that shows the aggregate active dispatch cores. Interestingly, gains are higher for dispatch, e.g., to achieve 450 KOp/s, Tailwind needs only 1/4 of dispatch cores used by RAMCloud.

Both observations confirm that, for updates, most of the resources are spent in processing replication requests. To get a better view on the impact when GET/PUT operations are mixed, we show Figure 4.12. It represents active worker and dispatch cores, respectively, when varying clients. When requests consist of updates only, Tailwind reduces worker cores utilization by 15% and dispatch core utilization by 50%. This is stems from the fact that a large fraction of dispatch load is due to replication requests in this case. With 50/50 reads and writes, worker utilization is slightly improved to 20% while it reaches 50% when the workload consists of 5% writes only.

Interestingly, dispatch utilization is not reduced when reducing the proportion of writes. With 5% writes, Tailwind utilizes even more dispatch than RAMCloud. This is actually a good sign, since read workloads are dispatch-bound. Therefore, Tailwind allows RAMCloud to process even more reads by accelerating write operations. This is implicitly shown in Figure 4.12 with "Replication" graphs that represent worker utilization due to waiting for replication requests. For update-only workloads, RAMCloud spends 80% of the worker cycles in replication. With 5% writes, RAMCloud spends 62% of worker cycles waiting for replication requests to complete against 49% with Tailwind. The worker load difference is spent on servicing read requests.

4.3.7 Scaling with Available Resources

We also investigated how Tailwind improves internal server parallelism (i.e. more cores). Figure 4.13 shows throughput and worker utilization with respect to available worker cores. Clients (fixed to 30) issue 50/50 reads and writes to 4 servers. We do not count the dispatch core with available cores, since it is always available. With a single worker core per machine, RAMCloud serves 430 KOp/s compared to 660 KOp/s for Tailwind with respectively 4.5 and 3.5 worker cores utilization. RAMCloud can over-allocate resources to avoid deadlocks, explaining why it can go above the core limit. Tailwind scales better when increasing the available worker cores. RAMCloud does not achieve more throughput with more than 5 available cores. Tailwind improves throughput up to all 7 available cores per machine.

Even though both RAMCloud and Tailwind exhibit a plateau, this is actually due to the dispatch thread limit that cannot take more requests in. This suggests that Tailwind allows RAMCloud to better take advantage of per-machine parallelism. In fact, by eliminating the replication requests from dispatch, Tailwind allows more client-issued requests in the system.

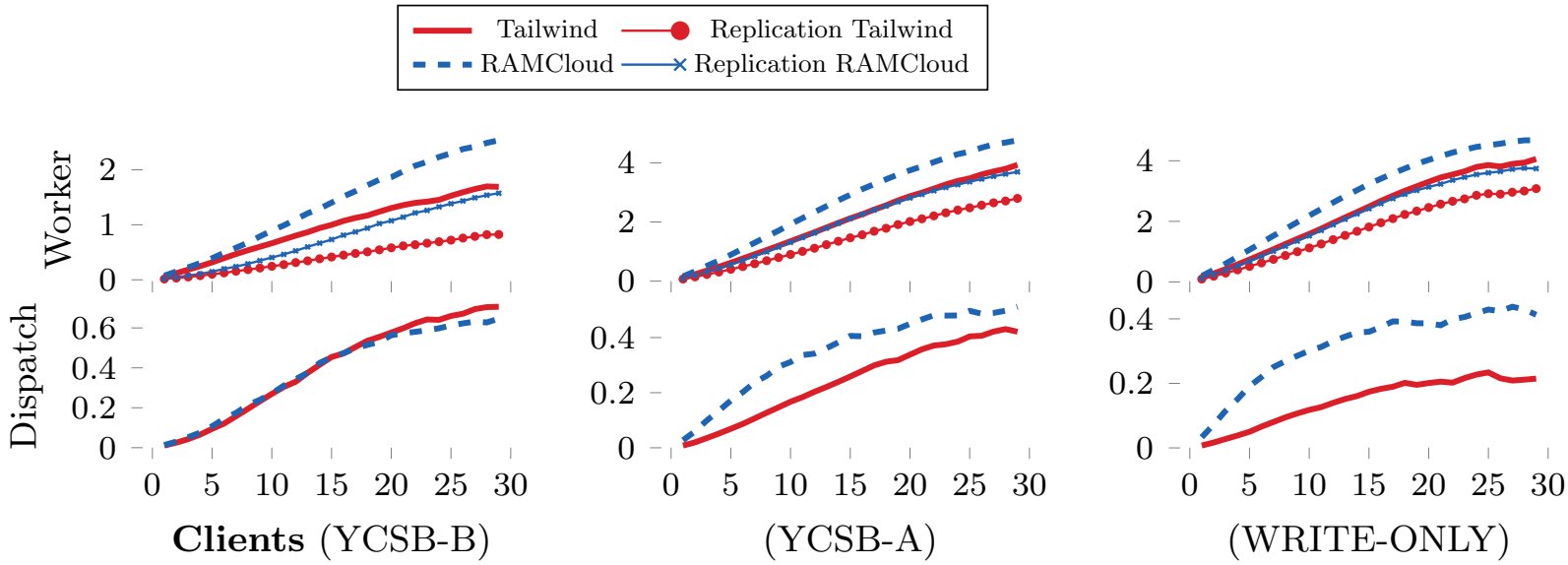


Figure 4.12 – Total dispatch and worker cores utilization per server in a 4-server cluster. "Replication" in worker graphs represent the fraction of worker load spent on processing replication requests on primary servers.

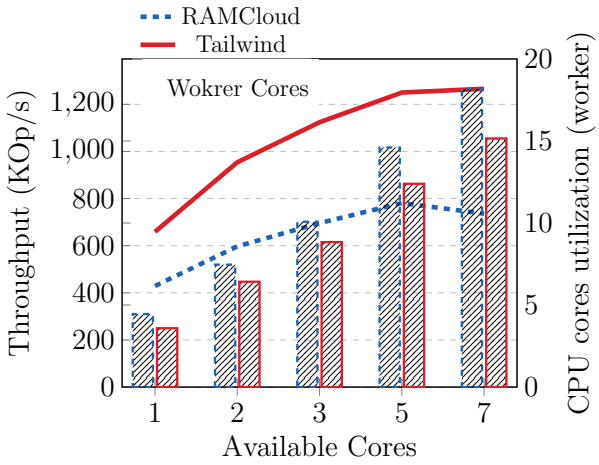


Figure 4.13 – Throughput (lines) and total worker cores (bars) as a function of available cores per machine. Values are aggregated over 4 servers.

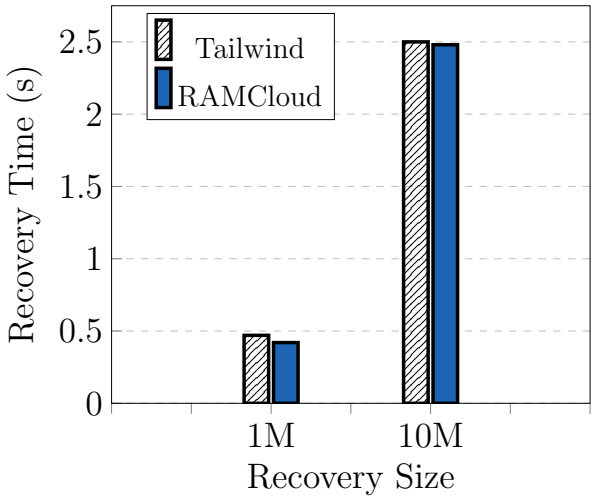


Figure 4.14 – Time to recover 1 and 10 million objects with 10 backups.

4.3.8 Impact on Crash Recovery

Tailwind aims to accelerate replication while keeping strong consistency guarantees and without impacting recovery performance. Figure 4.14 shows Tailwind's recovery performance against RAMCloud. In this setup data is inserted into a primary replica with possibility to replicate to 10 other backups. RAMCloud's random backup selection makes it so that all backups will end up with approximately equal share of backup data. After inserting all data, the primary kills itself, triggering crash recovery.

As expected, Tailwind almost introduces no overhead. For instance, to recover 1 million 100 B objects, it takes half a second for Tailwind and 0.48 s for RAMCloud. To recover 10 million 100 B objects, Both Tailwind and RAMCloud take roughly 2.5 s.

Tailwind must reconstruct metadata during recovery (§ 4.3.3.4), but this only accounts for a small fraction of the total work of recovery. Moreover, reconstructing metadata is only

necessary for `open` buffers, i.e. still in memory. This can be orders of magnitude faster than loading a buffer previously flushed on SSD, for example.

4.4 Discussion

4.4.1 Scalability

A current limitation of one-sided RDMAs is due to the requirement of reliable-connected QPs. Since information on QPs is cached at the NIC level, increasing the number of connections (e.g. scaling number of servers) over NIC cache size causes one-sided RDMA performance to collapse [39]. By targeting only replication requests, Tailwind is not subject to such a limitation; in most common storage systems data is partitioned into chunks that are replicated several times [11]. At any moment, only a small set of backups is replicated to.

4.4.2 Metadata Space Overhead

In its current implementation, Tailwind appends metadata after every write to guarantee RDMA writes atomicity (§ 4.3.1). Although this approach appears to introduce space overhead, RAMCloud's log-cleaning mechanism efficiently removes old checksums without performance impact [61].

A different implementation could consist in appending the checksum with log-backup data instead. This would completely remove space Tailwind space overheads. However, this would raise a challenge on backups: how to decide backup buffers size. In the current implementation primary DRAM log-storage and buffers on backups are the same. If checksums were appended to backup-data only, then buffers on backups will not have the same size as segments on primaries. A fixed-size buffer with only small objects will tend to have more checksums than a buffer which contains large objects, therefore more backup data could be stored on the latter.

In general, Tailwind adds only 4 bytes per object which is much smaller than, for example, RAMCloud headers (30 bytes).

4.4.3 Applicability

RDMA networks. Tailwind relies on RDMA-based networks to efficiently remove replication-CPU overheads. Historically, RDMA has been supported by Infiniband-based networks only [33, 58]. Recently, RDMA is supported in Ethernet in the form of RoCE or iWARP [31]. More importantly, recent studies suggest that RDMA-based networks [99, 55] are becoming common in modern datacenters.

Note that Tailwind improves clients end-to-end latency but does not require any change on their side. Tailwind only requires RDMA-based networks on the storage system side to operate.

Distributed logging. Distributed logging is widely used approach to redundancy [56, 44]. Many recent in-memory key-value stores achieve durability and availability through distributed logging as well [61, 21, 89, 63]. Tailwind leverages log-backup data-layout in order to ensure the atomicity of one-sided RDMA writes.

4.4.4 Limitations

Tailwind is designed to keep the same system-level consistency guarantees. It synchronously waits for the acknowledgement from the remote backups NICs in order to return to the client. Although it can do work in parallel, worker cores have to stall waiting until the QP generates a work completion in order to move to process another write request. For instance, Figure 4.5 shows that multiple updates can be queued and processed together. A better approach would be to pipeline replication requests. Worker cores would be able to perform some additional work and be notified when a work completion pops from the QP. RAMCloud's current RPC-based replication protocol is not pipelined neither [61] since its goal was to provide correctness and be able to reason about fault-handling.

4.5 Related Work

One-sided RDMA-based Systems. There is a wide range of systems recently introduced that leverage RDMA [21, 22, 40, 54, 82, 89]. For instance, many of them use RDMA for normal-case operations. Pilaf [54] implements client-lookup operations with one-sided RDMA reads. In contrast, with HERD [40] clients use one-sided RDMA writes to send GET and PUT requests to servers, that poll their receive RDMA buffers to process requests. In RFP [40] clients use RDMA writes to send requests, and RDMA reads to poll (remotely) replies.

Many systems also use one-sided RDMA for replication. For instance, FaRM [21, 22], HydraDB [89], and DARE [63] use one-sided RDMA writes to build a message-passing interface. Replication uses this interface to place messages in remote ring buffers. Servers have to poll these ring buffers in order to fetch messages, process them, and apply changes.

No system that uses RDMA writes for replication leaves the receiver CPU completely idle. Instead, the receiver must poll receive buffers and process requests, which defeats one-sided RDMA efficiency purposes. Tailwind frees the receiver from processing requests by directly placing data to its final storage location. Moreover, Tailwind does not sacrifice consistency to improve normal-case system performance.

4.6 Conclusion

Tailwind is the first replication protocol that fully exploits one-sided RDMA; it improves performance without sacrificing durability, availability, or consistency. Tailwind leaves backups unaware of RDMA writes as they happen, but it provides them with a protocol to rebuild metadata in case of failures. When implemented in RAMCloud, Tailwind substantially improves throughput and latency with only a small fraction of resources originally needed by RAMCloud.

With Tailwind, we make a step forward towards providing better efficiency in in-memory storage systems. As we show in Chapter 2, replication is expensive in in-memory storage systems, and it negatively impacts the energy efficiency and performance. Tailwind is a general design that can be applied to replicate data for any in-memory storage system.

In the next Chapter, we show how we leverage lessons learned from Chapter 3 and the design we present in this Chapter to build efficient replication and fault tolerance mechanisms for stream-storage systems.

Chapter 5

Optimizing Fault tolerance in Stream Storage: KerA

Contents

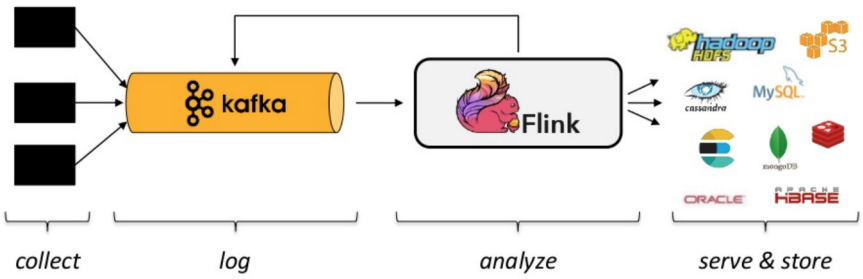


Figure 5.1 – A stream processing pipeline

It is usually taken that providing durability, strong-consistency (in the form of linearizability), and performance is very hard [38]. Especially for stream processing applications that require real-time data access.

In this chapter, we show how lessons learned in the last two chapters helped us providing consistency, durability, and performance in a stream store.

KerA [51] is a new low-latency stream store architectured and implemented within the framework of the PhD thesis of Ovidiu Marcu. It was designed to cope with existing static stream-ingestion systems by introducing a dynamic partitioning scheme that elastically adapts to the number of producers and consumers.

Next we give an overview of KerA and of its goals. Then we show how we enhanced KerA to augment it with consistency and durability. Specifically we describe the replication and crash recovery mechanisms we designed and integrated to KerA.

5.1 Stream Storage: Real-time Challenges

A stream is an unbounded and infinite set of events flowing from data sources (e.g. sensors) to a data processing layer. The main goal of a stream-processing platform is to extract value from data in real time. Typical applications include finance, fraud detection, scientific simulations, IoT, etc.

5.1.1 Stream Processing: An Overview

Typical stream-processing pipelines, as show in Figure 5.1, are composed of four major components:

- **Data Source:** Data sources correspond to any entity periodically generating data and sending it to the processing platform. They can be sensors, brokers which generate logs, devices, etc. The data source usually interfaces with the data ingestion layer.

- **Data Ingestion:** The second layer consists of the ingestion layer that collects and logs events produced by the data source. This layer is responsible of acquiring and providing an easy and efficient access API to data.

- **Stream Processing Engine (SPE):** a SPE is the core of the stream-processing platform as it consumes messages from the data ingestion and processes them. Because a stream is unbounded, the SPE maintains a "*state*" which allows it to rollback in case of failures.

- **Storage Layer:** The storage layer usually consists of a high-throughput cold storage, such as HDFS or Cassandra [76, 45]. It is not required to provide low latency, however, it is expected to provide high read throughput, typically for applications that would like to access historical data.

5.1.2 Stream Processing Ingestion and Storage Requirements

According to Stonebraker [79], there are some challenging requirements for real-time stream processing:

Keep the Data Moving. "To achieve low latency, a system must be able to perform message processing without having a costly storage operation in the critical processing path" [79]. As highlighted, the storage system can add a lot of unnecessary latency to the real-time processing process. Therefore, the system that is ingesting stream data should incur as less overhead as possible.

Integrate Stored and Streaming Data. Many stream processing applications must compare real-time data against historical. For instance, fraud detection applications have to compare *old* to *new* patterns to detect irregularities. Therefore, the ingestion system must *durably store* data and provide efficient access to it.

Guarantee Data Safety and Availability. Stream storage systems must provide high availability. For instance financial applications cannot afford to loose data and expect their application to run transparently and correctly all the time, even if a failure happen. Not only the stream processing needs to failover, but the ingestion system should be able to transparently hide failures.

Partition and Scale Applications Automatically. Stream processing applications rely on distributed commodity machines to transparently scale applications. Not only must scalability be provided easily over any number of machines, but the resulting application should automatically and **transparently** load-balance over the available machines, so that the application does not get bogged down by a single overloaded machine [79].

In order to meet these requirements many stream-storage systems have been developed.

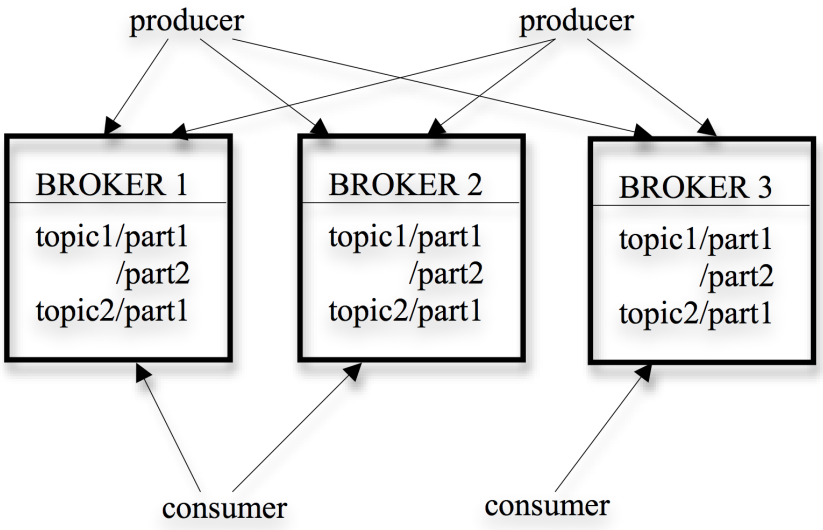


Figure 5.2 – Kafka architecture

5.1.3 Existing Solutions for Stream Storage

5.1.3.1 Apache Kafka

Kafka is a messaging-based log aggregator. In Kafka, a stream of messages of a particular type is defined by a topic. A producer can publish messages to a topic. The published messages are then stored at a set of servers called *brokers*. A consumer can subscribe to one or more topics from the brokers, and consume the subscribed messages by pulling data from the brokers.

A Kafka cluster typically consists of multiple brokers. To balance load, a topic is divided into multiple partitions and each broker stores one or more of those partitions. Multiple producers and consumers can publish and retrieve messages at the same time.

Each partition in Kafka corresponds to a log. Physically, a log is implemented as a set of segment files. Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For better performance, we flush the segment files to disk only after a configurable number of messages have been published or a certain amount of time has elapsed. A message is only exposed to the consumers after it is flushed [44].

Recently, Kafka was augmented with the ability to provide exactly-once semantics through transactions [38]. This means that even if a producer retries sending a message, it leads to the message being delivered exactly once to the end consumer. Exactly-once semantics is the most desirable guarantee in distributed systems [38]. However, this comes at a high performance cost: measurements made by Kafka developers report that Kafka's throughput and latency can degrade from 15% to 225% when enabling transactional support [44].

5.1.3.2 Apache DistributedLog

DistributedLog [28] is a replicated log stream store built on top of Bookkeeper [35]. In DistributedLog, a log stream is segmented into multiple log segments where each single of them is stored as sequence of entries called *ledgers*. Ledgers are append-only and data written to them cannot be modified.

A major difference with Kafka, is that all the records of a log stream are sequenced by the owner of the stream. Writers are interfaced with proxies. Therefore, all writes go through a central entity serializing all messages and assign sequence numbers to every single message. To read from DistributedLog, readers provide a sequence number.

To guarantee total ordering, DistributedLog only allows one active log segment accepting writes at any time. All writes are replicated to a configurable quorum of nodes. Writes are only acknowledged once replicas have persisted them on disk.

DistributedLog relies on proxies to serialize reads and writes [28]. By doing so it ensures write-ordering among partitions, at the expense of throughput, as proxies reduce parallelism.

5.1.3.3 Apache Pulsar

Usually, messaging services can be divided in tow categories: queuing or streaming [81]. Queuing is a point-to-point messaging abstraction where ordering is not important. Streaming means messages are strictly ordered and there can be only one consumer per topic/partition.

Pulsar [67] unifies the two messaging paradigms. Pulsar achieves queuing by deploying a shared subscription with round-robin delivery. While increasing parallelism usually requires increasing the number of partitions, Pulsar allows applications to scale active consumers beyond the number of partitions. Moreover, Pulsar also implements streaming by deploying an exclusive subscription with ordered, sequential delivery. This allows applications to process messages in order and offers the possible to perform catch reads.

Pulsar can achieve low latency, however, it has a message-based interface. For instance, to read fresh data, a reader must consume all messages until reaching the most recent data. This can significantly impede applications that need to access specific offsets of a stream.

5.1.4 Limitations of Existing Solutions

Many of these systems offer good and predictable performance in normal cases. However, some of them trade some features for performance. For instance, Kafka is not usually deployed in synchronous mode, i.e., replicas do not have to be in-sync in order to reply to users requests. Many deployments report issues related to replicas falling behind and slow replication overall [93, 37].

Systems like DistributedLog are designed for geo-replicated deployments. Although they offer high availability and strict guarantees, operating latencies of these systems are still very high compared to the latencies we are targeting [28].

More generally we find the following shortcoming in state-of-the-art stream-storage systems:

Elasticity. Elasticity is one of the most important aspects of stream storage. Since streams are unbounded and infinite, their arrival rate might vary throughout time. Therefore a stream-storage system must be able to dynamically adapt to the producers/consumers rate transparently. Systems like Kafka do not provide such a feature, however they provide overall good performance.

Availability. As we have shown in the last two chapters, replication brings a lot of overhead in all kind of storage systems. Although systems such as DistributedLog, which is one of the fastest strongly-consistent stream-storage systems, offer predictable latencies even under high loads, they don't offer latencies for applications we envision for KerA such as fraud detection and financial applications or autonomous vehicles. For instance, with exactly-once semantics recently introduced, Kafka performs transactions in the order of 100 ms [38], whereas autonomous vehicles require latencies less than 50 ms. More precisely, DistributedLog relies on Bookkeeper as the underlying storage layer and adds serialization layers on top of it in the forms of write and read proxies. These layers add considerable latency as all clients requests have to go through a single point to write to the system. Moreover, every write has to be replicated, and committed to disk on replicas in order to be available to the readers. This implies latencies of at least tens of milliseconds, as reported in [28].

Strong Consistency and Exactly-Once Semantics. Linearizability is the strongest form of consistency for concurrent systems. Providing linearizability is usually equivalent to guaranteeing exactly-once delivery of messages [46]. Kafka recently introduced the exactly once feature [38] based on transactions. However, as reported in [38], compared with deployments without exactly-once feature, the performance dramatically decreases as soon as it is enabled. This is natural in the sense that transactions in practise require extensive usage of timeouts and message exchanges. On the other hand, DistributedLog provides exactly-once, however it can do so by limiting the rate at which messages are exposed to consumers. Indeed, they require first a message to be serialized by a proxy writer, so that it gets timestamped. Second, it has to be replicated to a quorum of replicas. Third, replicas have to commit to disk the writes and only then acknowledge the write. Only after all acknowledgments are received that the recently written data can be exposed to clients.

5.2 Background on KerA

We chose KerA as it has a promising approach for scalability and performance compared to the state-of-the-art. It builds atop RAMCloud's RPCs [61], which offer low-latency by leveraging kernel-bypass. Moreover, KerA has fine grain stream partitioning compared to the state-of-the-art, which allows higher parallelism and therefore, more throughput for stream processing applications.

5.2.1 Data Partitioning Model

KerA implements dynamic partitioning based on the concept of *streamlet*. Streamlets are similar to Kafka partitions, as each stream is composed of a fixed number of streamlets. One

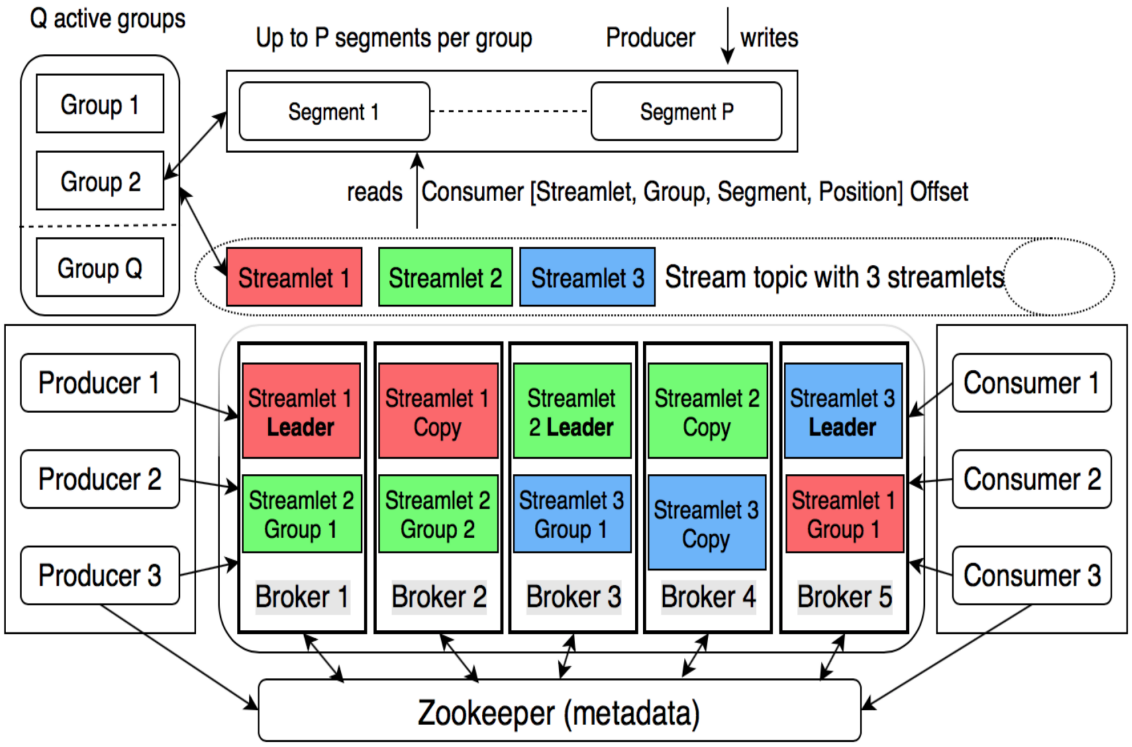


Figure 5.3 – KerA architecture [51]

could imagine a cluster that starts with a single node having M streamlets. KerA allows the stream to scale out to M nodes for horizontal scalability.

At the level of a single node, KerA introduces more levels of granularity: *groups* and *segments*. A streamlet is composed of an unbounded numbers of groups. A group contains a fixed number of segments. To control the level of streamlet parallelism allowed on each broker,a limited number of groups can be active at a given moment. Elasticity is achieved by assigning an initial number of brokers $N \geq 1$ to hold the streamlets M, $M \geq N$. As more producers and consumers access the streamlets, more brokers can be added up to M. In order to ensure ordering semantics, each streamlet dynamically creates groups and segments that have unique, monotonically increasing identifiers. Brokers expose this information through RPCs to consumers that create an application offset defined as [streamId, streamletId, groupId, segmentId, position] based on which they issue RPCs to pull data. The position is the physical offset at which a record can be found in a segment. The consumer initializes it to 0 (broker understands to iterate to first record available in that segment) and the broker responds with the last record position for each new request, so the consumer can update its latest offset to start a future request with.

Stream records are appended in order to the segments of a group, without associating an offset, which reduces the storage and processing overhead. Each consumer exclusively processes one group of segments. Once the segments of a group are filled (the number of segments per group is configurable), a new one is created and the old group is closed (i.e., no longer enables appends). A group can also be closed after a timeout if it was not appended in this time [51].

5.2.2 KerA Architecture

A typical cluster of KerA comprises a set of nodes each of which will host a KerA broker and a KerA backup. Similarly to RAMCloud (Figure 3.1), these services are interlinked with a centralized metadata broker called *coordinator*. Typically the coordinator is replicated through a logging system, such as Zookeeper (Figure 5.2). The coordinator has a high level view of the cluster. It keeps the location of streams and streamlets. Producers and consumers directly contact the coordinator to retrieve information about the location of streams and their streamlets. This information is cached, and only retrieved if there is a configuration change in the cluster. The coordinator is also responsible of managing recovery as we describe later in this chapter.

5.2.3 Clients

5.2.3.1 Producers

When a producer is created, it is assigned a unique identifier. To be able to interact with KerA, a producer only needs to know about streamlets present in the stream. To do so KerA provides with an API that enables producers to query stream-related information.

After a producer gets relevant information on the stream, it can start producing records and send them to KerA brokers. Producers can choose to route requests either in round-robin or by key. A key is important as it helps KerA in routing requests to a specific machine. In stream storage systems, it is common to use routing keys to reason about related events [65]. Usually a key can be a "date" or a "machine-id".

Usually producers can be sensors, devices, or brokers generating logs, periodically sending data to KerA. Therefore, they have the ability to batch records before sending them. In Kafka for instance, batching is important for the overall system performance [44]. Similarly, in KerA producers can batch records in two forms: either fix a timeout t after which a producer sends all accumulated records; or fix a size s of batches, and after each s bytes accumulated, send the request.

5.2.3.2 Consumers

Consumers consist usually of stream-processing engines that will periodically pull data. Similarly to producers, each consumer will have a unique identifier in the system. KerA exposes an API allowing consumers to discover groups in streamlets. After a consumer discovered groups, it can start requesting records by sending a "*getNextSegment*" RPC to a KerA broker. The RPC has to provide the stream-id, the *streamlet-id*, the *group-id*, the last *segment-id* read, the *offset* from which the consumer wants to start reading data; and *maximum-data size* that the readers wants. KerA will iterate over batch entries and append up to *maximum-data size* bytes to the response RPC. KerA will also append the offset of the last batch entry that was read. It is the responsibility of the consumers to keep manage the last offset read, as in subsequent read operations, they will have to specify that offset to move forward.

Although applications can be parallel and have many tiny consumers, they will share the same identifier. Usually, stream-processing engines will use many data sources threads to concurrently pull data. However, these data sources do not necessarily communicate to

share their sate. KerA handles this by only allowing a single consumer per group. Therefore, multiple consumers sharing the same identifier can safely pull data from KerA without worrying about data duplication. In contrast, systems such as Kafka solve this issue by limiting the number of consumers sharing the same identifier to a single one per partition, which limits applications performance.

5.2.4 Current Limitations of KerA

KerA as presented in [51] is mainly focusing on stream partitioning and not on fault-tolerance. It could achieve good performance for producers and consumers by allowing fine grain data access.

In order to enable strongly-consistent replication in current KerA's design, we would have to restrict the number of active groups in every KerA broker, which completely nullifies KerA's benefits.

Furthermore, for applications that need durability and availability, it is complicated to enforce consistent replication with multiple open segments in a single log. The problem is that having multiple open segments creates a lot of complications when it comes to replication and crash recovery. For instance, in RAMCloud, a server's log could only have a single open segment at a time for several reasons: (1) during recovery, a backup only needs to peek into the last open segment of a master to discover all segments (through the segment digest [61]) of that master; (2) if log data is stored in an open segment while other segments which precede it in the log are still open, the data may not be detected as lost if it is lost; this is because if all the replicas for the segment with the data in it are lost the coordinator will still conclude it has recovered the entire log since it was able to find an open segment; (3) when performing asynchronous writes, many segments might be allocated without the data in them being durably replicated; for instance, if segments are opened out-of-order, then the log digests might indicate segments which have not made it yet to the backups.

5.3 Our Contributions to KerA

We presented some of the notions introduced by KerA in [51]. Because we want to provide strong consistency and performance at the same time, we had to re-design the storage part in KerA. Specifically, we introduce a new log-storage abstraction we call *MultiLog* that allows parallel appends and replication up to a certain threshold. The MultiLog allows reasoning about consistency, which was not possible in the original KerA's implementation. Moreover, we took advantage of RAMCloud's fast crash recovery mechanism and augmented KerA with it to enable availability.

5.3.1 Replicating Logs

Log-structured storage naturally fits the streaming data model. As data flows from producers to the storage over time, it is easier to reason about order through a log-structured approach. In KerA, we leverage the RAMCloud log-structured storage abstraction as a building block for the in-memory storage layer.

In KerA, a log physically stores multiple segments. Each time a segment is created, a configurable number of backup replicas are chosen to replicate that segment to. At start up,

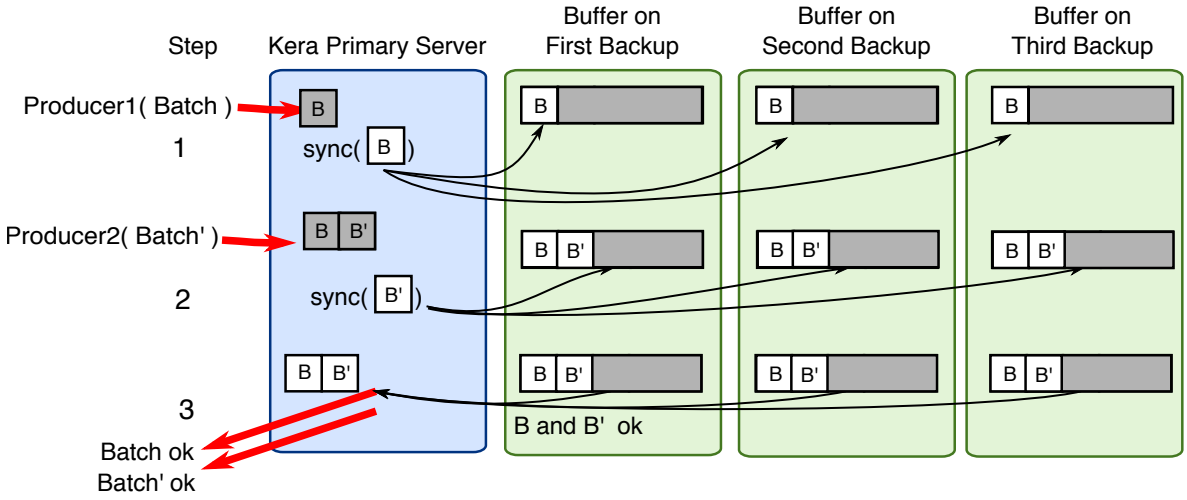


Figure 5.4 – Replication of a single log in KerA, similar to RAMCloud [61]. A log can perform concurrent appends, however, a single thread is allowed to perform sync at a time.

every backup will create a pool of in-memory buffers in order to store backup data. These buffers are periodically flushed to secondary storage. This approach allows replication to go faster by avoiding the slow disk I/Os. Once a buffer has been flushed to secondary storage, its memory space can be reclaimed and used to store backup data for a new segment.

Next, we will describe the replication mechanisms in KerA. We first start with the replication process of a single log, then we will describe how we can extend the same principles to the MultiLog in a single broker.

5.3.1.1 Replicating A Single Log

We describe the basic log replication in Figure 5.6 in the following steps:

- Step 1: A primary KerA broker receives a batch from Producer 1. The broker first writes the batch B to memory, then initiates three replication RPCs to three replicas.
- Step 2: In the meantime, the same broker received in parallel a request from a different producer, that was mapped to the same log. Since the request was received after the batch B has been written to memory, then the thread processing the request can acquire the append-lock and write the batch B'.
- Step 3: If appending B' is fast enough, it can be replicated alongside batch B with the same request. As soon as the broker receives the acknowledgement from backups, it can acknowledge the appends to the producers.

Whenever replicating a segment, a KerA broker will provide the following information to the backcup: (1) broker identifier; (2) stream-id; (3) streamlet-id; (4) group-id; (5) segment-id; (6) log-id. Although streamlets, groups and segments identifiers might be the same in different streams, stream identifiers are unique in the system. Therefore, the combination of these 5 fields make it possible to uniquely identify a segment in the system.

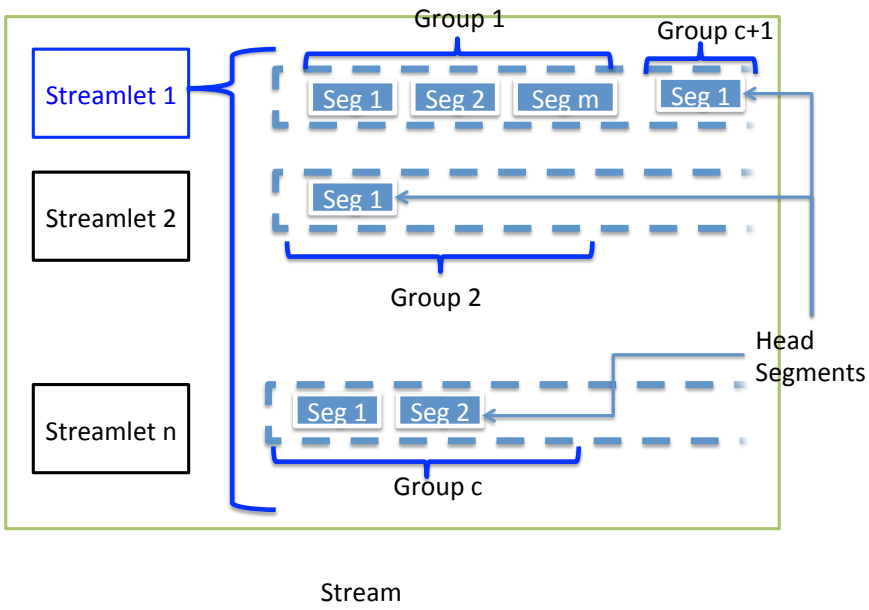


Figure 5.5 – KerA MultiLog abstraction. Every dotted line corresponds to a log. A log may contain different groups and segments. The log is decoupled from the stream representation and serves as a storage abstraction.

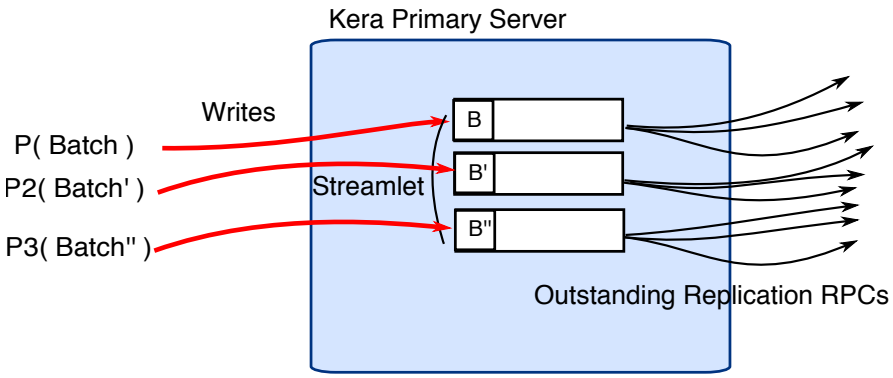


Figure 5.6 – Replication of MultiLogs in KerA. As each log is self-managed, multiple append requests can be replicated concurrently.

5.3.1.2 Replicating a MultiLog

A KerA broker supports an arbitrary number of streamlets. This allows a very flexible deployments as we showed in § 5.2.1. However, in production deployments, we envision that each broker would store a single streamlet. Based on this assumption, we designed a specific abstraction to store streams in KerA: the *MultiLog* abstraction.

Bascially, we represent every streamlet as a MultiLog which consists of Q logs as show in Figure 5.5. Consequently, each log in a MultiLog stores an increasing number of groups and segments. At any point in time, a log has a head segment to which it can append.

Performance Considerations. A MultiLog allows Q parallel appends to a streamlet. In contrast with DistributedLog, for example, KerA allows much larger number of producers to append concurrently to the stream without synchronization need.

Consistency Considerations. Consistency was the most important factor that led us to consider the MultiLog abstraction. In fact, without such an abstraction, it would not be possible to allow multiple concurrent producers at the same time while providing high performance. As shown in Figure 5.6, a single log abstraction does not allow multiple appends concurrently. They have to synchronize, otherwise consistency would not be guaranteed.

Fault Tolerance. Similarly to RAMCloud, KerA will choose a replica for each segment. For now, we use a default random replica selection strategy. This leads to a uniform segment distribution across the cluster. In case of failures, KerA leverages segment distribution in order to recover data from a large number of backups.

5.3.1.3 Optimizing Producers

We modified the producers to heavily rely on batching. Batching impacts KerA in many ways. But essentially, KerA will try to create a unique batch entry per append request received. For instance, a producer might send x number of records in a request, which will be wrapped and considered as a single entry in KerA. To enable this, we created a new entry type called *batch*. The main purpose of a batch entry is to reduce the overhead induce by iterating over multiple entries during append operations, which increases considerably the append performance.

When requests reach a KerA broker, they are deterministically routed to a log. This is made possible since each record is meant for a particular streamlet of a stream. In addition to these two pieces of information, a KerA broker will map every producer to a certain log in a streamlet by using the producer identifier. For instance, in a very simple scenario where there are Q producers sending requests to a KerA broker having a single streamlet with Q logs, the broker will be able to process concurrently all requests from the Q producers without blocking. When the number of producers grows, requests from producers that get mapped to the same log can be processed concurrently, but threads servicing these requests will have to synchronize over an *append-lock* proper to each log.

5.3.2 Crash Recovery

Fault tolerance is one of the most important aspects of KerA. Data producers and stream application must not be disrupted during failures. KerA particularly targets use-cases where strong consistency is required. For instance, Goldman Sachs uses a Kafka cluster in order to ingest more than 1 Tb a week. As reported by their infrastructure team, a machine failure happens at least 5 times a year[91]. In case of failures, they report that access to the topics affected by failures will cease. Moreover, they would have to manually add replicas to the failed partition in order. Similarly, Pinterest reports that their Kafka cluster might experience multiple failures a day [36]. In case of failures, they report that they have to manually execute Kafka scripts in order to replace dead brokers.

To cope with these limitations, KerA recovery is based upon the RAMCloud fast crash recovery mechanism. KerA will always guarantee consistent access to data. Therefore, it does not trade availability for consistency. However, it leverages all backups of the cluster in order to recovery as fast as possible from failures.

Next we present the crash recovery mechanism we use in KerA, which is an adaptation of the crash recovery mechanism of RAMCloud [61].

5.3.2.1 Detecting Failures

Servers periodically send random ping RPCs to each other to check if they are alive. A ping RPC is an application level RPC as opposed to ICMP pings. Therefore, if broker receives a responses to a ping RPC, it means that the KerA service is still running on the remote broker. If the receiver of a ping RPC fails to respond on time then the sender will suspect a crash on the receiver. As a result, it notifies the coordinator, which will try in its turn to send a ping RPC again the suspected KerA broker. If the suspected broker responds the coordinator will not take action. Otherwise, if the broker does times out again, the coordinator will consider the broker crashed and initiate the crash recovery mechanism.

5.3.2.2 Locating Segments of a Broker

The coordinator has only a partial view of the cluster. It knows where streams and streamlets are stored. However, it does not keep information on the location of backup replicas. Therefore, to retrieve this information, it has to query all backups of the cluster. To start a recovery, a coordinator will first send an RPC to all backup brokers in order to collect information on the location of the the failed broker's backups.

Once replicas receive the query, they will respond with all segments they store for the crashed KerA broker. When the coordinator receives the responses from all backups, it will first check whether it could find all segments of the crashed master. A key enabler of this operation is the presence of a *log digest* on each segment. A log digest is just a list of all segments appended so far to the log. Therefore, a backup has only to look into the segment currently in-memory in order to retrieve the most recent digest for a certain log. The digest information is sent as a response to the query from the coordinator.

Note that unlike RAMCloud, the coordinator has to handle not a single, but many logs for a crashed broker. Typically, a crashed broker could have up to QxM logs. However, because segments are scattered across all the backups, it is unlikely to end up with incomplete logs.

After the coordinator is able to find a list of valid segments composing the crashed master, it will have to distributed streamlets replay across masters. Currently, we simply map each streamlet to a new broker in a round-robin fashion. The intuition is to be able to parallelize streamlets reconstruction while avoiding to overwhelm brokers that might be already accessed by producers and consumers in parallel.

5.3.2.3 Reconstructing Streamlets

Each broker participating to recovery will receive the list of streamlets it has to recover and the location of the backups it must get data from. As soon, as a broker receives a recovery request from the coordinator, it starts fetching data from the backups.

In parallel, backups will respond to the queries from brokers and load all relevant backup data, as disk speed is a bottleneck. Servers will request segments in the order they appeared on the crashed master. This order is important, especially for streaming applications.

Servers will also have to re-replicate segments as they are reconstructed to guarantee durability of the recovered streamlets. Once all brokers have replayed all of the segments of all streamlets, they contact the coordinator. When the coordinator receives a notification from a broker that has finished recovering a streamlet it sends a special RPC to order the broker to take ownership of that streamlet and start servicing requests again. The broker will then be able to broker consumers and producers right away.

After all broker finished the recovery work, and correctly re-replicated all streamlets, the coordinator will send an RPC to all backups, ordering the removal of old-replicas from memory and disk.

5.4 Evaluation

5.4.1 Implementation Details

The current prototype results in approximately 3500 lines of code on top of KerA's initial version, written in C++. Although the current prototype supports TCP connectivity, we mainly focused on optimizing the kernel-bypass version of the transport protocol which runs on Infiniband networks. The main motivation behind that is that we ported Tailwind in KerA, which can only be possible in Infiniband-based networks for now. Although Ethernet-based RDMA implementations already exist, they do not perform well [99, 58]. Regardless of that, we are currently working on the integration of an Ethernet-based kernel-bypass transport which, we hope, can deliver performance which is closer to the Infiniband-based transport. However, we would not support RDMA-based replication in that case.

5.4.2 Experimental Setup

Experiments were done on up to 20 nodes from the Dell r320 cluster (Table 4.1) on the CloudLab [70] testbed. We implemented producers and consumers similarly to Kafka [44]. Producers are configured with a total number of objects to send uniformly to all brokers. We limit the total number of objects at any point of time as a function of the available memory on each producer. Currently, producers are pipelined, i.e., they send up to a configurable number of RPCs. Periodically, a producer will check completed RPCs and immediately launch new RPCs.

Consumers on the other hand are configured to pull up to a total number of objects. They discover streams, and segments from the coordinator through KerA's API. Usually a consumer will discover a segment, and pull objects until (1) there are no objects left to pull; (2) it finished reading the segments and pulls the next segments. We configure consumers to continuously pull objects until they consumed all objects (i.e. all objects written by producers).

We configured every broker to assign Q logs per streamlet, where Q is equal to the number of available cores per machine.

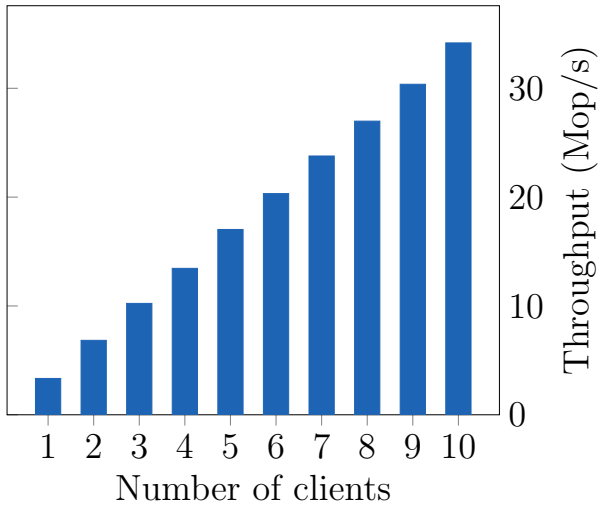


Figure 5.7 – Scalability of a single KerA broker while running concurrent producers. For each experiment, producers send 40 million objects.

5.4.3 Peak Performance

We first take a look at the peak performance a single KerA broker can sustain while multiple producers send a total of 40 million objects in parallel.

Figure 5.7 shows the throughput of write operations in millions of operations/second for a single KerA broker. Producers are configured to batch and send up to 100 objects/request. We can see that a single client can sustain a throughput of 3 million writes per second. Increasing the number of producers does not affect their respective throughput: the broker can scale linearly with the incoming load. For instance with 10 producers, the broker can sustain a load of 33 million writes per second, which is perfectly linear.

This experiment shows the effectiveness of the MultiLog abstraction. In KerA, every producer request is deterministically mapped to a specific log by using the unique producer identifiers. By doing so, KerA allows multiple producer requests to be processed in parallel without the need for synchronization. Therefore, KerA can linearly scale with increasing concurrent write requests.

To further illustrate the peak performance of a broker, in Figure 5.8 we fix 10 producers and vary the batch size. Intuitively, batching more objects per request results in better throughput, at the expense of the latency. The Figure shows that indeed batching can help in increasing performance. However, the maximum throughput that can be achieved in a single broker is reached with a batch size of 20 KObjects, where a broker can sustain a load of 40 million writes per second. This is roughly equivalent to 4 GB/s, or 30Gbps, which is enough throughput to saturate most of nowadays commodity machines NICs.

5.4.4 Integrating Tailwind in KerA

We ported Tailwind in KerA as it naturally fits the log-based storage we use to store stream data. We did not make any change to the original Tailwind implementation, which shows that it can be easily ported to other systems.

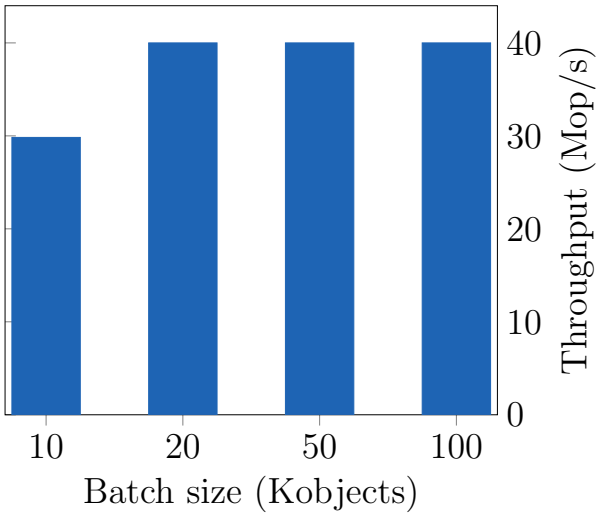


Figure 5.8 – Scalability of a single KerA broker while running concurrent 10 Producers. For each experiment, producers send 40 million objects with different batch sizes

To illustrate the benefits of RDMA-based replication we show Figure 5.9 that represents the throughput over time of a 4-node cluster when 6 producers run concurrently sending a total of 20 million objects. We run the RPC-based replication and the RDMA-based replication and vary the replication factor in both cases.

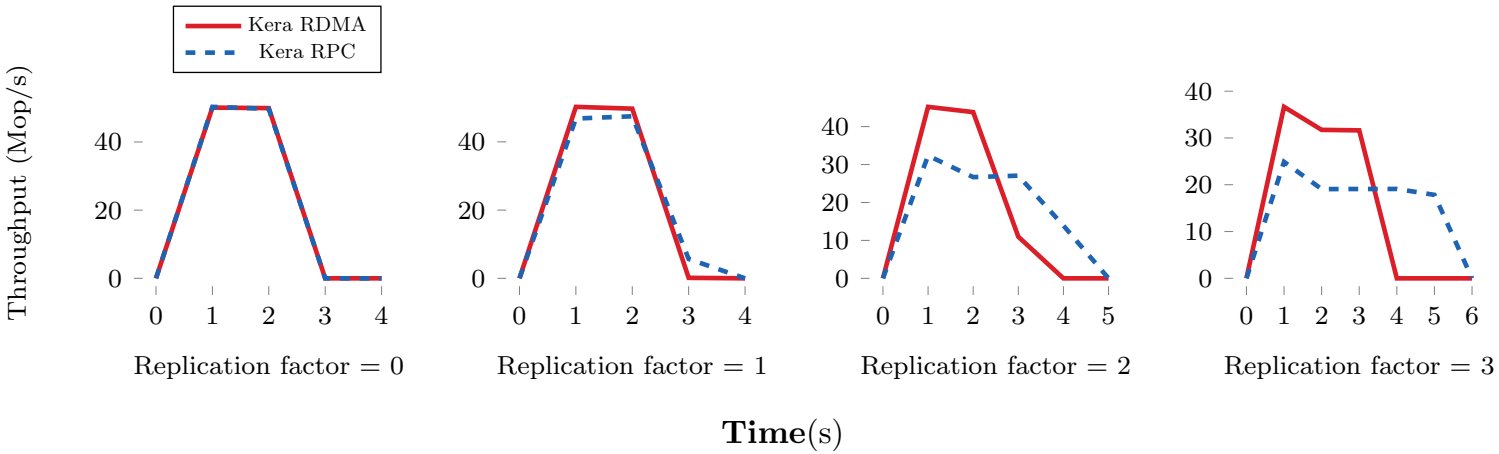


Figure 5.9 – Throughput of a 4-node cluster, while running 6 concurrent producers sending 20 million objects each. Each graph corresponds to a different replication factor.

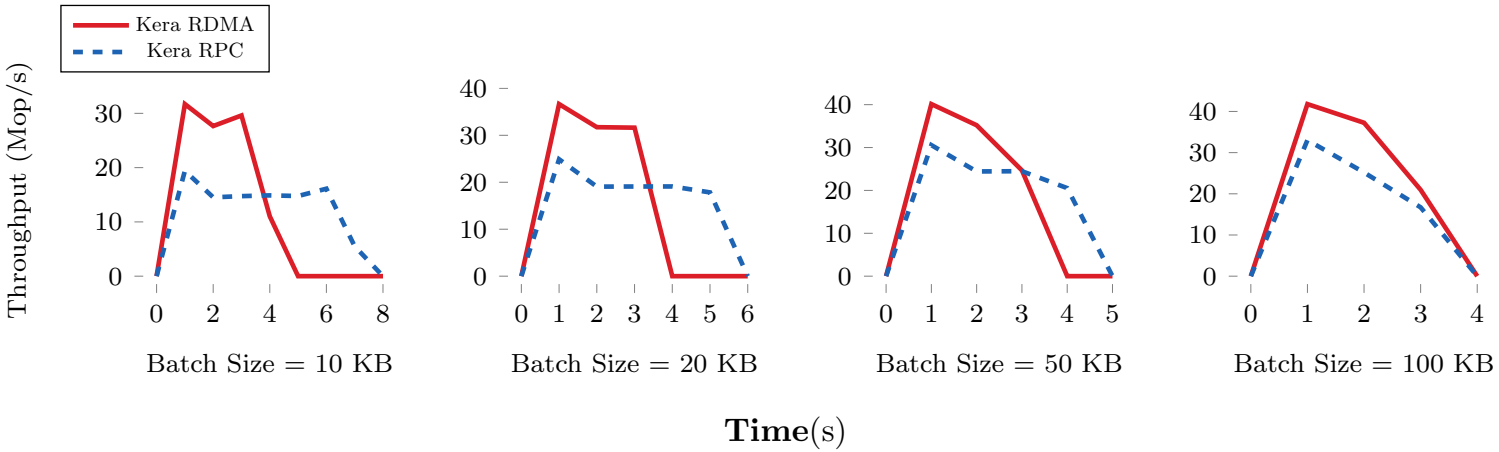


Figure 5.10 – Throughput of a 4-node cluster with replication factor of 3, while running 6 concurrent producers sending 20 million objects each. Each graph corresponds to a different batch size.

We can see that with replication factors of 0 and 1 the performance are almost equal, capping at 47 million writes per second. When increasing to a replication factor of 2 and 3, the throughput of KerA with RPC-based replication drops to 30 and 20 millions writes per seconds, respectively. With RDMA-based replication, however, the throughput still peaks at 45 and 39 millions writes per second with replication factors of 2 and 3, respectively.

These results stem from the fact that with higher replication factors, brokers will have to process a larger number of replication requests. Leveraging RDMA for replication enables brokers (which act as backup at the same time) to focus their CPU on processing write requests rather than processing replication requests, which explains the gain in throughput.

We also investigated the impact of RDMA with respect to the size of requests sent by the producers. Figure 5.10 shows the throughput with the same configuration as above, with only difference that we fix the replication factor to 3 and in this case we vary the batch size.

With smaller batch sizes, we can see that the throughput is clearly better when using RDMA-based replication. For instance, with batch sizes of 10 KB and 20 KB, KerA with RDMA-based replication can sustain a throughput of 30 and 35 million writes per second, respectively; in contrast, with RPC-based replication, KerA can only sustain a throughput of 20 and 25 millions writes per second, respectively. As expected, when increasing the batch size, up to 100 KB, the gab between RDMA and RPC-based replications gets smaller. However, even with a batch size of 100 KB, with RDMA-based replication KerA can deliver a peak throughput of 40 million writes per second, while it can only achieve 30 million writes per second with RPC-based replication.

This experiment confirms the benefit we can get from enabling RDMA-based replication in KerA with respect to the throughput. While batching is known to increase throughput, we've shown that even with less batching, KerA can sustain a high throughput. This can be very appealing for producers that have limited amount of memory such as sensors. In this case, by leveraging RDMA we can accelerate the rate at which producer requests are processed, without requiring more batching to achieve higher throughput.

5.4.5 Understanding the Efficiency of Replication in KerA

To understand why Tailwind improves the performance in KerA, we investigate the CPU with RPC and RDMA-based replication. More precisely, we look at the dispatch and worker cores utilization under different configurations.

Figure 5.11 shows the aggregated CPU cores utilization of a 4-node KerA cluster of both dispatch and worker cores. We run 15 producers and produce a total of 40 million objects uniformly distributed across all brokers. More Specifically, Figure 5.11a, the dispatch core utilization when varying the replication factor from 0 to 3. We can see that with RPC-based replication the dispatch utilization increased from 0.1 with replication factor 0 to 0.9 with replication factor 3, which corresponds to 25% of utilization for a single node. On the other hand, with Tailwind, the dispatch utilization increased from 0.1 to 0.4 with replication factor 3, which is more than 50% decrease compared to using RPCs.

These results stem from the fact that when KerA brokers use RDMA to replicate, their dispatch cores are freed from handling replication requests.

Figure 5.11a shows the worker utilization cores, i.e., cores that are responsible of performing writes and replicate those writes to backups. We can see from the Figure that RPC-based

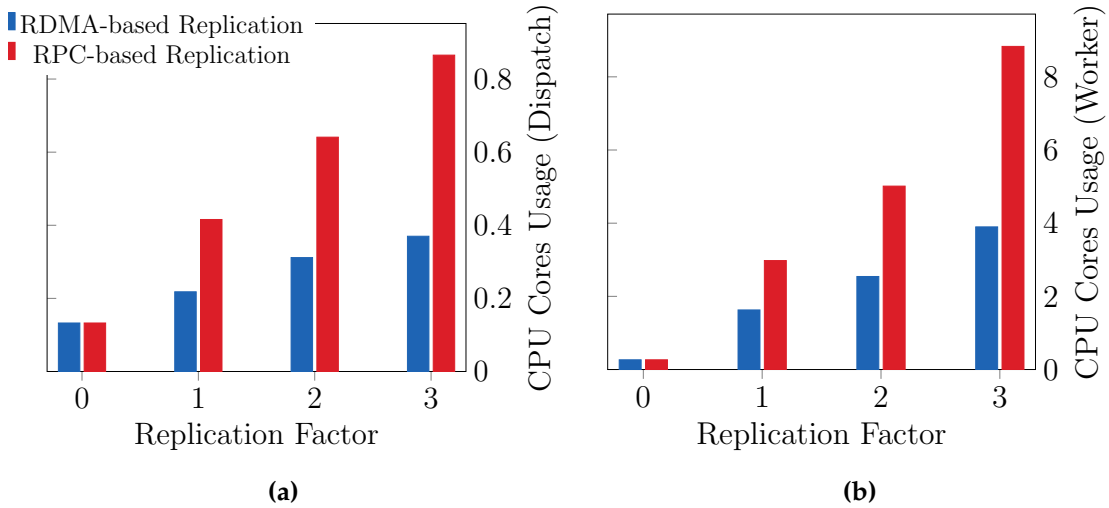


Figure 5.11 – Aggregated (a) dispatch and (b) worker cores utilization over a 4-node KerA cluster when using RPC and RDMA-based replication. We run 15 producer sending a total of 40 million objects.

replication leads to a 0.2 worker utilization with replication factor of 0 and up to 9 worker cores with a replication factor of 3. On the other hand, with Tailwind, KerA uses up to 4 worker cores with a replication factor of 3, which is more than 50% decrease compared to using RPCs.

These results are consistent with the dispatch utilization results. Specifically, when using Tailwind, KerA brokers do not use their worker cores to process replication requests, which leads to substantial CPU cycles savings.

5.4.6 The Impact of the MultiLog Abstraction

Finally, to show the benefit of the MultiLog abstraction, we look at the impact of concurrency, when varying the number of available logs per broker. This is equivalent to comparing the original KerA implementation (which had a single log) and our implementation.

Figure 5.12 shows the total throughput of a single KerA broker when varying the number of logs. We fix the number of producers to 3 with a fixed batch size of 100 KB. Note that for this experiment only, we used the c6220 Cluster from CloudLab. We can see from the figure that with a single log, which corresponds to the original KerA implementation with a single active group per partition, a broker can sustain up to 28 million appends/s. When increasing the number of available logs, we can see a clear increase in the throughput: with 2 logs a broker sustains 35 million appends/s, up to 55 million appends/s with 4 logs.

This stems from the fact that with more producers than available logs, there is a synchronization cost. Indeed, a thread handling a producer's append request has to acquire an append lock in order to make progress in processing the request. This is why increasing the number of logs leads to a better throughput.

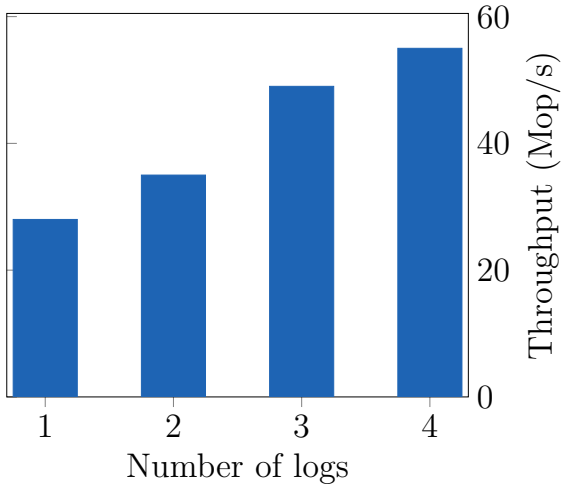


Figure 5.12 – Total throughput of a single KerA broker when varying the number of logs in a MultiLog. A single log corresponds to the original KerA implementation. The number of producers is fixed to 3 with a batch size of 100 KB

5.5 Discussion

Although the current implemented is still at the state of a prototype, and some features are still not fully implemented, the results we have a very promising and let us envision many optimizations for the future. For instance, we envision to develop and RDMA-based API for clients. This would greatly benefit some applications such as HPC simulations that run on dedicated platforms. However, the drawback is that pushing complexity to the clients library can compromise ease of use and constrain KerA users.

However, there are some limitations to the MultiLog abstraction. Specifically, some applications require very large numbers of streamlets. For instance, Netflix deploys on average 10K partitions per stream, which translates to 10K streamlets in our case. Having such a number of streamlets requires large amount of memory on backups: for every streamlet there will be $Q \times R$ open segments on backups, where Q is the number of logs and R is the replication factor. With 1 MB segments and 8 logs per streamlets and a replication factor of 3, storing 10K streamlets would require 80 GB of DRAM storage on each backup to store the open segments.

5.6 Conclusion

In this chapter we present how we designed and implemented replication and fault tolerance in the KerA stream store. We first describe how we model and store streams to fully leverage a machine's multicore architecture. Then we describe the MultiLog parallel replication. Finally, we present how we leverage RAMCloud's recovery mechanism and adapt it in KerA.

KerA provides linearazability, and can ingest data up to a rate of 1GB/s in a commodity machine, durably. Key enabling factors are: leveraging parallelism; avoiding network over-

heads through kernel-bypass; eliminating replication CPU overhead on backups by using RDMA-based replication.

Chapter 6

Conclusion

Contents

The emergence of data intensive and real-time applications has created new challenges for storage system designers and service providers. These applications not only need fast data access, but they often require consistency and data durability. In-memory storage systems have played a key role in enabling fast data access. More recently, persistent in-memory storage systems can completely eliminate the need for multiple storage layers by providing mechanisms to persist data, while guaranteeing fast and consistent data access. To be able to achieve that, these systems have mechanisms that use aggressively most available resources, especially CPU and memory. Moreover, their fault-tolerance mechanisms aim at providing maximum availability and durability. This usually translates in additional resource usage during operations such as replication or crash recovery.

On the other hand, efficient-datacenter usage has become one of the major goals for major service providers. The cost of powering server systems has been steadily rising with higher performing systems. There are both economic and ecological incentives to better utilize datacenter servers and avoid resource ”waste”.

In this thesis, we have showed that there is a wide gap to fill to make in-memory storage systems more efficient and reliable. By studying their performance and energy efficiency, we show that durability mechanisms, such as replication, can be quite expensive in terms of energy and that they can negatively impact performance. Therefore, We proposed a new CPU-efficient replication mechanism based on RDMA. Finally, we leveraged our contributions and apply them in the context of stream storage by designing and implementing efficient replication and fault tolerance mechanisms for a state-of-the-art stream store.

6.1 Achievements

Hereafter, we summarize the achievements obtained in each of the three contributions of the thesis.

6.1.1 Studying Performance vs Energy Tradeoff in Modern In-memory Storage Systems

In-memory storage systems are widely used in the form of distributed caching and in-memory processing. Yet, we find very little work on studying the tradeoff between performance and energy efficiency in in-memory systems, even though, it is well-known that DRAM contributes to a large fraction of a server's power consumption. By means of experimental evaluation, we have tried to shed the light on the performance vs energy trade-off in in-memory storage. Specifically, we chose RAMCloud — a state-of-the-art in-memory key-value store — and evaluated it with different workloads under various configurations.

We reveal that although RAMCloud scales linearly in throughput for read-only applications, it has a non-proportional power consumption. We also show that under heavy concurrent writes, RAMCloud's performance can significantly drop. We relate both observations to the thread-management mechanism in RAMCloud which is optimized for latency, but not for throughput.

We also show that replication can be a major bottleneck for performance and energy, especially in CPU-bound configurations. The cause is that since in RAMCloud server can act both as primary and secondary, replication contends with clients read and write operations. We find that this phenomenon also extends to the crash recovery phase, when data needs to be replayed and re-replicated.

6.1.2 Design and Implementation of an Efficient RDMA-based Replication Protocol

As we have shown, replication can introduce large overheads in in-memory storage systems, especially in strongly-consistent ones. Specifically, replication puts a lot of pressure on servers CPU, which leads to a contention between replication requests and clients requests on a server's CPU.

To mitigate this, we propose Tailwind, an RDMA-based replication protocol that leaves backups completely idle during replication. A major challenge that arise when using RDMA, is that receivers are unaware of data being written to their DRAM. In the event of primary-replica failures, this could lead to data corruption, as backups have no means to identify

valid parts of backup data. Tailwind tackles this issue by providing backups with a recovery protocol that allows them to reconstruct metadata information about backup data in case of failures.

We implemented Tailwind in RAMCloud, and show that it can dramatically reduce replication overheads. For instance, it can reduce CPU cores usage up to 4x while improving the throughput of the system by 70%.

6.1.3 Design and Implementation of Efficient Durability and Availability Techniques for Fast Stream Storage

The advent of data-intensive applications and real-time stream processing led to unprecedented needs in terms of storage. More specifically, recent applications, such as self-driving cars, not only require fast data access, but they also need persistence and consistency.

By leveraging lessons learned from our first-two contributions, we have designed and implemented replication and fault tolerance mechanisms in a state-of-the-art stream store, named Kera [51]. A major challenge in stream storage is how to provide exactly-once semantics while guaranteeing availability and high performance.

By leveraging recent techniques such as kernel-bypass, RDMA-based replication, and multicore architectures, we show that we are able to provide unprecedented levels of performance. For instance, a single kera server running on a commodity machine can ingest up to a rate of 1GB/s durably.

6.2 Perspectives

6.2.1 A general purpose fault tolerance mechanism

Tailwind is a protocol to replicate, safely, with one-sided RDMA. We envision to build a general purpose strongly-consistent RDMA-based replication system with a very simple API to persist data. It can be seen as an alternative to RAID, but orders of magnitude faster (throughput and latency). We argue that this can be possible, especially with the advent of new technologies such as NVRAM.

Although, there are many challenges in adopting such kind of system. For instance, a key challenge is how users interact with the system. Specifically, does this system only provide backup services, or should we allow reads as well. Another question is how and where to recover data in case of failures. A simple solution would be to have some predefined strategies among which a sysadmin could chose when deploying my system. Luckily, we can leverage techniques already used in open-source software community, such as RAMCloud in order to help with these challenges.

There are more specific challenges related to this idea.

Leveraging NVRAM for Persistence

DRAM is volatile. To persist data, one must eventually store it in a persistent medium. Nowadays, systems such as RAMCloud or FaRM persist data in disk and flash storage. The issue is that they still have to perform slow I/O to persist data, but also when they

recover data after failures. We believe that leveraging RDMA to replicate data to NVRAM is a plausible solution. Not only it would eliminate the need to perform slow I/O in backups, but it would also save CPU cycles. However, there are issues such as enforcing cache flush to the NVRAM, which is not trivial to achieve in an efficient way nowadays.

How to Efficiently Replicate?

A major challenge is CPU efficiency in strongly-consistent systems. During replication, a sender has to actively wait for the acknowledgement from the remote NIC before notifying the client, which wastes CPU cycles. Performing a context switch is too costly in low-latency systems (A durable write takes 10 μs to complete in RAMCloud). Therefore, we believe recent systems such as Arachne [3], from Stanford, could help in performing low-cost context switches. This would not only make replication fast, but also CPU efficient by allowing threads to execute other tasks while data is transferred to remote backups.

6.2.2 Continuous Availability in Stream Storage

When a failure happens, in a strongly-consistent environment, it is not possible to resume service, for lost data, right away. One has to wait until a recovery (failover) finishes, until service can be *safely* resumed. The reason is that, in an environment where there are $f + 1$ replicas, one must get the acknowledgement of a quorum of f replicas in order to consider an operation successful.

However, recent applications, such as stream-processing applications have real-time processing needs, which implicate that data should be *always* available. Many service providers actually settle for lower consistency guarantees to provide higher availability. But some applications, such as fraud detection do not tolerate data inconsistency.

Although it is seems impossible to keep service up and running if one the replicas fail, we think it might be possible if we are dealing with append-only data. The rationale is that during a failures, only f (or less) out of $f + 1$ replicas are available. Therefore, it is impossible to get a quorum of f backups acknowledging, for example, a write request. Simply because on full replica has to be reconstructed. However, we make assumptions that would let a system provide continuous service even under failures. The first assumption is that recovery must be an atomic process. Either it fully recovers lost data or it fails, it can't recover partial data. The second assumption is to separate the ownership of data appended *after* a crash happens. Therefore, if a subsequent crash happens in one of the f replicas, it wouldn't affect data created after the initial crash, which is why this protocol can only work for append-only data.

We could go further and tolerate to serve data with only a quorum of $f - 1$ backups if a failure happens such as Bookkeeper does [35]. In this case we could provide continuous availability even for readers. However, there would be numerous challenges to deal with in order to make this happen. Specifically, we would have to develop mechanisms to guarantee exactly-once semantics in a different that is currently done in Kera.

Résumé en Français

Contexte

Depuis plusieurs décennies, les données ont été essentiellement utilisées par les entreprises à des fins commerciales. Toutefois, de nos jours, les données occupent une place de plus en plus importante dans notre vie quotidienne. Les technologies émergentes, telles que les smartphones, les voitures autonomes et les assistants personnels sont en train de façonner notre mode de vie et la manière dont nous interagissons avec autrui. Il faut noter aussi que les applications basées sur ces technologies génèrent des quantités de données de plus en plus importantes. Les réseaux sociaux et les sites d'e-commerce, par exemple, Facebook et Amazon, sont parmi les applications les plus importantes en popularité mais aussi en terme de volume de données généré. Ceci s'explique par le fait que ces applications sont facilement accessibles, ce qui a entraîné une adoption massive de la part de milliards d'utilisateurs. Par exemple Facebook rapporte 2 milliards d'utilisateurs actifs et 8 milliards de téléchargements de vidéos par jour. Avec un tel volume de données générées, ces applications posent de nouveaux défis en matière de stockage et de traitement de données. Durant la dernière décennie le problème majeur était de faire face à ces larges volumes de données, toutefois, aujourd'hui les défis principaux sont : comment stocker, traiter, et extraire de la valeur de ces larges volumes de données en temps réel ?

Du point de vue de la technologie de stockage, les systèmes de stockage basés sur des disques durs (HDD) ont longtemps perduré dans l'écosystème du Big Data. Néanmoins, ces derniers n'arrivent plus à suivre la cadence en terme de besoin en performances (rapidité du temps d'accès au système de stockage). Toutefois, avec la croissance rapide des mémoires vives (DRAM), les fournisseurs de services se basent de plus en plus sur ce type de technologies qui offrent des temps d'accès nettement plus rapides que les HDD. Récemment, une nouvelle famille de systèmes de stockage basée sur la DRAM est apparue, avec pour but d'exploiter au mieux les mémoires vives. Cette famille de systèmes garantit la disponibilité des données et la resilience face aux pannes, éliminant au passage le besoin d'autres couches de stockage, qui garantissait la disponibilité de données et la tolérance aux pannes, pouvant affecter la rapidité d'accès aux données.

La DRAM est volatile, c'est-à-dire que sans alimentation électrique, elle perd les données stockées. De ce fait, les systèmes de stockage distribués en mémoire utilisent tout de

même des HDD afin de sauvegarder les données. Ces systèmes sont, pour la plupart, équipés de mécanismes permettant de garantir la disponibilité des données en cas de pannes, et sont particulièrement optimisés pour ce cas. Ils doivent souvent lire les données des disques durs en cas de panne, ces derniers étant des ordres de grandeur moins rapides que les memoires vives. Ces optimisations ont plusieurs répercussions : (1) Les systèmes de stockage en mémoire doivent créer des données redondantes, typiquement en replicant, ce qui induit un surcoût en matière de performance ; (2) Pour garantir la disponible de données en cas de pannes, la plupart des systèmes ont des mécanismes de recouvrement utilisant les ressources de manière agressive. Par exemple, le système de stockage en mémoire RAMCloud fait en sorte que toutes les machines du cluster participent au processus de recouvrement en cas de panne d'un seul noeud, afin de recréer les données le plus rapidement possible.

Étant donné que la DRAM est nettement plus chère comparée à d'autres supports de stockage traditionnellement utilisés, tels que les HDD, les architectes ont particulièrement optimisé les systèmes en mémoire en matière de performance, se focalisant moins sur d'autres aspects, tels que la consommation énergique, ou l'efficacité, c'est-à-dire le nombre de cycles CPU dépensés par opération. Bien que les technologies de stockage Flash et les réseaux, aient tous deux connu des avancées impressionnantes dans leurs vitesses et capacités respectives, les fréquences des CPU ont stagné. Ceci est principalement dû à des limites physiques en matière de finesse de gravure des transistors. La loi de Moore avait prédit que le nombre de transistors dans les puces doublerait chaque deux ans : bien que ce fût vrai de 1975 à 2012, les constructeurs se sont récemment heurtés aux frontières quantiques, et ne peuvent plus améliorer la fréquence des processeurs. De ce fait, les CPU sont devenus peu à peu les goulots d'étranglement dans le chemin d'accès aux données.

Il apparaît donc, que les systèmes de stockage en DRAM vont : (1) induire des coûts supplémentaires en matière de consommation énergétique ; (2) Il sera de moins en moins intéressant de co-localiser des applications et les systèmes de stockage en mémoire sur les mêmes machines, qui, il faut le rappeler, fut un de leurs principaux buts. Bien qu'il y ait eu beaucoup d'efforts dans l'espoir d'optimiser les performances des systèmes de stockage en mémoire, nous avons décidé de nous intéresser à l'aspect tolérance aux pannes. Dans ce manuscrit, nous montrons qu'il existe un manque à combler en matière d'efficacité de systèmes de stockage en mémoire, en étudiant les performances et l'efficacité énergétique de ces derniers. Ensuite, nous démontrons qu'actuellement les mécanismes garantissant la resilience des données, tels que la réplication, peuvent se révéler inefficaces et monopolisent trop de ressources. Pour résoudre ces problèmes, nous proposons un nouveau mécanisme de réplication, que nous appelons Tailwind, tirant profit des nouvelles technologies réseaux. Enfin, utilisant de nos études et techniques, nous nous intéressons à un cas concret de système de stockage temps réel. En effet, nous démontrons qu'il est possible, avec nos contributions, d'apporter des performances, de garantir la cohérence forte et la résilience de données dans les systèmes de stockage pour le stream processing, ceci étant considéré comme particulièrement difficile de nos jours.

Cette thèse fut réalisée dans le cade du projet européen Big Storage, avec comme but de caractériser les surcoûts des mécanismes garantissant la tolérance aux pannes dans les systèmes de stockage en mémoire et pour optimiser ces mécanismes afin d'améliorer les performances et augmenter l'efficacité de ces systèmes.

Contributions

Nous resumons les principales contribution de cette thèse comme suit :

Étude de l'Efficacité des Systèmes de Stockage en Mémoire

La plupart des applications Web populaires, telles que Facebook et Twitter, conservent une grande partie de leurs données dans la mémoire vive afin de garantir un accès rapide aux données. Alors que les travaux antérieurs visaient à exploiter la faible latence de l'accès en mémoire à grande échelle, la comprehension de l'efficacité des systèmes de stockage en mémoire est très limitée. Par exemple, aucun travail n'a examiné le coût des opérations en termes de cycles de processeur ou de consommation d'énergie, alors même que l'on sait que la mémoire vive constitue un goulot d'étranglement énergétique fondamental dans les infrastructures actuelles (la DRAM consomme jusqu'à 40% de la consommation électrique d'un serveur). Dans cette contribution, par le biais d'une évaluation expérimentale, nous avons étudié les performances et l'efficacité de RAMCloud, un système de stockage en mémoire connu. Nous nous concentrons sur des métriques telles que la performance, les cycles de CPU par opération et la consommation d'énergie. Nous révélons une consommation d'énergie non proportionnelle due au mécanisme de threading. Nous constatons également que le schéma de réplication actuel implémenté dans RAMCloud limite les performances et entraîne une consommation d'énergie élevée. Nous montrons que la réplication peut jouer un rôle négatif dans la rapidité de récupération de données après une panne, dans certaines configurations.

Tailwind : Réplication Rapide et Atomique Basée sur RDMA

La réplication est essentielle pour la tolérance aux pannes. Cependant, dans les systèmes en mémoire, il s'agit d'une source de surcoût en performance élevée, comme nous l'avons montré dans la première contribution. L'accès direct à la mémoire distant (RDMA) est un moyen attrayant de créer des copies redondantes de données, car celui-ci a une faible latence et évite l'utilisation du CPU. Cependant, les approches existantes entraînent toujours une copie de données redondante et des récepteurs actifs. Dans les approches existantes, les récepteurs assurent le transfert de données atomique en vérifiant et en n'appliquant que les messages entièrement reçus. Nous proposons Tailwind, un protocole de réplication sans copie pour les bases de données en mémoire. Tailwind est le premier protocole de réplication qui élimine toutes les copies de données et qui contourne complètement les processeurs des recepteurs, les laissant ainsi les inactives. Tailwind garantit que toutes les écritures sont atomiques en exploitant un protocole qui détecte les transferts RDMA incomplets. Tailwind améliore considérablement le débit de réplication et la latence par rapport à la réplication conventionnelle basée sur les appels de procédures distants (RPC). Dans les systèmes symétriques où les serveurs servent des requêtes et agissent comme des répliques, Tailwind améliore également le débit normal en libérant les ressources du processeur du serveur pour le traitement des requêtes. Nous avons implémenté et évalué Tailwind sur RAMCloud. Notre évaluation montre que Tailwind améliore jusqu'a 1.7x le débit de traitement des requêtes normales de RAMCloud. Il réduit également les latences médianes et les 99 pourcentilles de 2x et 3x, respectivement.

Amélioration des Performances, de la Durabilité et de la Cohérence dans les Systèmes de Stockage pour Streaming

Alors que Les nouvelles applications generant de larges volumes de données, les analyses et les simulations scientifiques, présentent des besoins sans précédent en termes de performances de stockage. Elles présentent des exigences similaires : un modèle de données basé sur les flux, une faible latence et un haut débit, et dans de nombreux cas, elles nécessitent une cohérence forte de données. Cependant, les solutions existantes, telles qu'Apache Kafka, ne sont pas optimisées en termes de disponibilité et de cohérence. Dans cette contribution, nous présentons la conception et la mise en oeuvre de mécanismes de durabilité dans KerA, un système de stockage pour streaming à latence faible. KerA fournit un accès aux données a granularité fine, permettant un débit élevé tout en respectant la charge des clients. En outre, Kera permet un service de données continu, même en cas de défaillance, sans sacrifier la cohérence de données.

Publications

Publications dans des Conférences Internationales

- **Yacine Taleb**, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. *Tailwind : Fast and Atomic RDMA-based Replication*. In Proceedings of the 2018 USENIX Annual Technical Conference, (**USENIX ATC'18**). July 2018, Boston, United States.
- **Yacine Taleb**, Shadi Ibrahim, Gabriel Antoniu, and Toni Cortes. *Characterizing Performance and Energy-efficiency of The RAMCloud Storage System*. In Proceedings of the The 37th IEEE International Conference on Distributed Computing Systems (**ICDCS'17**). May 2017, Atlanta, United States.

Publications dans des Workshops Internationaux

- **Yacine Taleb**. *Optimizing Fault-Tolerance in In-memory Storage Systems*, in the EuroSys 2018 Doctoral Workshop (**EuroDW'18**). April 2018, Porto, Portugal.
- **Yacine Taleb**, Shadi Ibrahim, Gabriel Antoniu, Toni Cortes. *An Empirical Evaluation of How The Network Impacts The Performance and Energy Efficiency in RAMCloud*, Workshop on the Integration of Extreme Scale Computing and Big Data Management and Analytics in conjunction with IEEE/ACM **CCGRID'17**. May 2017, Madrid, Spain.
- **Y. Taleb**, S. Ibrahim, G. Antoniu., T. Cortes. *Understanding how the network impacts performance and energy-efficiency in the RAMCloud storage system*. Big Data Analytics : Challenges and Opportunities, held in conjuction with ACM/IEEE **SC'16**. November 2016, Salt Lake City, United States.

Bibliography

[1] URL: www.grid5000.fr/.

[2] Hrishikesh Amur, James Cipar, Varun Gupta, et al. "Robust and flexible power-proportional storage". In: *ACM Symposium on Cloud Computing (SoCC '10)*. 2010, pp. 217–228.

[3] *Arachne*. Stanford University, 2018. URL: https://github.com/PlatformLab/Arachne (visited on 2018-08-03).

[4] *Architecture of Giants: Data Stacks at Facebook, Netflix, Airbnb, and Pinterest*. https://blog.keen.io/architecture-of-giants-data-stacks-at-facebook-netflix-airbnb-and-pinterest-9b7cd881af54. (Visited on 2018-06-29).

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, et al. "Workload Analysis of a Large-scale Key-value Store". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '12. London, England, UK: ACM, 2012, pp. 53–64. ISBN: 978-1-4503-1097-0. DOI: 10.1145/2254756.2254766. URL: http://doi.acm.org/10.1145/2254756.2254766.

[6] Luiz André Barroso. "The Price of Performance". In: *Queue* 3.7 (Sept. 2005), pp. 48–53. ISSN: 1542-7730. DOI: 10.1145/1095408.1095420. URL: http://doi.acm.org/10.1145/1095408.1095420.

[7] Luiz Barroso, Mike Marty, David Patterson, et al. "Attack of the Killer Microseconds". In: *Commun. ACM* 60.4 (Mar. 2017), pp. 48–54. ISSN: 0001-0782. DOI: 10.1145/3015146. URL: http://doi.acm.org/10.1145/3015146.

[8] Eric Brewer. "Pushing the CAP: Strategies for Consistency and Availability". In: *Computer* 45.2 (Feb. 2012), pp. 23–29. ISSN: 0018-9162. DOI: 10.1109/MC.2012.37. URL: https://doi.org/10.1109/MC.2012.37.

[9] Nathan Bronson, Zach Amsden, George Cabrera, et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 49–60. ISBN: 978-1-931971-01-0. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson.

[10] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 24.1 (2005), pp. 18–28.

[11] Asaf Cidon, Stephen Rumble, Ryan Stutsman, et al. "Copysets: Reducing the Frequency of Data Loss in Cloud Storage". In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA, 2013, pp. 37–48. ISBN: 978-1-931971-01-0.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0.

[13] *Apache CoucheDB*. Apache, 2018. URL: http://couchdb.apache.org/ (visited on 2018-06-14).

[14] *CPU – The New Bottleneck?* 2016. URL: https://www.datacore.com/blog/cpu-the-new-bottleneck/ (visited on 2018-06-14).

[15] *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*. 2009. URL: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 2018-06-14).

[16] *Why Haven't CPU Clock Speeds Increased in the Last Few Years?* 2014. URL: https://www.comsol.com/blogs/havent-cpu-clock-speeds-increased-last-years/ (visited on 2018-06-14).

[17] Edward M. Curry. "Message-Oriented Middleware". In: *Middleware for Communications*. John Wiley and Sons, 2004, pp. 1–28. DOI: 10.1002/0470862084.ch1.

[18] P. Lake and P. Crowther. "Concise Guide to Databases". In: *Topics in Computer Science*. 2013, pp. 21–40. DOI: 10.1007/978-1-4471-5601-7_2.

[19] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM (CACM)* 51.1 (2008), pp. 107–113.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

[21] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, et al. "FaRM: Fast Remote Memory". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 401–414. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616486.

[22] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, et al. "No Compromises: Distributed Transactions with Consistency, Availability, and Performance". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 54–70. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815425. URL: http://doi.acm.org/10.1145/2815400.2815425.

[23] *Data Center Efficiency Assessment*. 2014. URL: https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf (visited on 2018-06-14).

[24] *IEEE 802: 400 Gb/s Ethernet Task Force*. 2018. URL: http://www.ieee802.org/3/bs/ (visited on 2018-06-14).

[25] Bin Fan, David G. Andersen, and Michael Kaminsky. "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL, 2013, pp. 371–384. ISBN: 978-1-931971-00-3.

[26] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. "Power Provisioning for a Warehouse-sized Computer". In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ISCA '07. San Diego, California, USA: ACM, 2007, pp. 13–23. ISBN: 978-1-59593-706-3. DOI: 10.1145/1250662.1250665. URL: http://doi.acm.org/10.1145/1250662.1250665.

[27] R. Patgiri and A. Ahmed. "Big Data: The V's of the Game Changer Paradigm". In: *IEEE International Conference on High Performance Computing and Communications (HPCC '16)*. 2016, pp. 17–24.

[28] S. Guo, R. Dhamankar, and L. Stewart. "DistributedLog: A High Performance Replicated Log Service". In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. Apr. 2017, pp. 1183–1194. DOI: 10.1109/ICDE.2017.163.

[29] *SAP HANA*. 2018. URL: https://www.sap.com/products/hana.html (visited on 2018-06-14).

[30] *SAP HANA sales fly but there's more to the in-memory story*. 2013. URL: https://www.zdnet.com/article/sap-hana-sales-fly-but-theres-more-to-the-in-memory-story/ (visited on 2018-06-14).

[31] *IB Specification Vol 1*. Release-1.3. InfiniBand Trade Association. Mar. 2015.

[32] *Internet Trends*. Kleiner Perkins, 2018. URL: https://www.slideshare.net/kleinerperkins/internet-trends-report-2018-99574140 (visited on 2018-06-14).

[33] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, et al. "High performance MPI-2 one-sided communication over InfiniBand". In: *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.* Apr. 2004, pp. 531–538. DOI: 10.1109/CCGrid.2004.1336648.

[34] Ryan Johnson, Ippokratis Pandis, Radu Stoica, et al. "Scalability of write-ahead logging on multicore and multisocket hardware". In: *The VLDB Journal* 21.2 (Apr. 2012), pp. 239–263.

[35] Flavio P. Junqueira, Ivan Kelly, and Benjamin Reed. "Durability with BookKeeper". In: *SIGOPS Oper. Syst. Rev.* 47.1 (Jan. 2013), pp. 9–15. ISSN: 0163-5980. DOI: 10.1145/2433140.2433144. URL: http://doi.acm.org/10.1145/2433140.2433144.

[36] *Kafka cluster healing and workload balancing*. https://medium.com/@Pinterest_Engineering/open-sourcing-doctorkafka-kafka-cluster-healing-and-workload-balancing-e51ad25b6b17.

[37] *[Kafka-users] Replicas Falling Behind*. 2018. URL: http://grokbase.com/t/kafka/users/153dbpbktn/high-replica-max-lag (visited on 2018-06-26).

[38] *Exactly-once Semantics are Possible: Here's How Kafka Does it*. 2018. URL: https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/ (visited on 2018-06-14).

[39] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-sided (RDMA) Datagram RPCs". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 185–201. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026892.

[40] Anuj Kalia, Michael Kaminsky, and David G. Andersen. "Using RDMA Efficiently for Key-value Services". In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Aug. 2014), pp. 295–306. ISSN: 0146-4833. DOI: 10.1145/2740070.2626299. URL: http://doi.acm.org/10.1145/2740070.2626299.

[41] Robert Kallman, Hideaki Kimura, Jonathan Natkins, et al. "H-store: A High-performance, Distributed Main Memory Transaction Processing System". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1496–1499. ISSN: 2150-8097. DOI: 10.14778/1454159.1454211. URL: http://dx.doi.org/10.14778/1454159.1454211.

[42] Rini T. Kaushik and Milind Bhandarkar. "GreenHDFS: Towards an energy-conserving, storage-efficient, hybrid Hadoop compute cluster". In: *USENIX International Conference on Power Aware Computing and Systems (HotPower '10)*. 2010, pp. 1–9.

[43] *Kernel Bypass*. 2017. URL: https://www.netronome.com/blog/avoid-kernel-bypass-in-your-network-infrastructure/ (visited on 2018-06-14).

[44] J. Kreps, N. Narkhede, and J. Rao. "Kafka: A distributed messaging system for log processing". In: *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*. 2011.

[45] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: http://doi.acm.org/10.1145/1773912.1773922.

[46] Collin Lee, Seo Jin Park, Ankita Kejriwal, et al. "Implementing Linearizability at Large Scale and Low Latency". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: ACM, 2015, pp. 71–86. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815416. URL: http://doi.acm.org/10.1145/2815400.2815416.

[47] Bojie Li, Zhenyuan Ruan, Wencong Xiao, et al. "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 137–152. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132756. URL: http://doi.acm.org/10.1145/3132747.3132756.

[48] Haoyuan Li, Ali Ghodsi, Matei Zaharia, et al. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: ACM, 2014, 6:1–6:15. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670985. URL: http://doi.acm.org/10.1145/2670979.2670985.

[49] Hyeontaek Lim, Dongsu Han, David G. Andersen, et al. "MICA: A Holistic Approach to Fast In-memory Key-value Storage". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 429–444. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616488.

[50] Aqdas Malik, Aqdas Malik, Kari Hiekkanen, et al. "Impact of privacy, trust and user activity on intentions to share Facebook photos". In: *Journal of Information, Communication and Ethics in Society* 14.4 (2016), pp. 364–382.

[51] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, et al. "KerA: Scalable Data Ingestion for Stream Processing". In: *ICDCS 2018 - 38th IEEE International Conference on Distributed Computing Systems*. Vienna, Austria: IEEE, July 2018, pp. 1–6. URL: https://hal.inria.fr/hal-01773799.

[52] *Memcached*. 2018. URL: https://memcached.org/ (visited on 2018-06-14).

[53] *Memory Prices from 1957 to 2017*. 2017. URL: https://jcmit.net/memoryprice.htm (visited on 2018-06-14).

[54] Christopher Mitchell, Yifeng Geng, and Jinyang Li. "Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store". In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA, 2013, pp. 103–114. ISBN: 978-1-931971-01-0.

[55] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, et al. "TIMELY: RTT-based Congestion Control for the Datacenter". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 537–550. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787510. URL: http://doi.acm.org/10.1145/2785956.2787510.

[56] C. Mohan, Don Haderle, Bruce Lindsay, et al. "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging". In: *ACM Trans. Database Syst.* 17.1 (Mar. 1992), pp. 94–162. ISSN: 0362-5915. DOI: 10.1145/128765.128770. URL: http://doi.acm.org/10.1145/128765.128770.

[57] *MongoDB*. 2018. URL: https://www.mongodb.com/ (visited on 2018-06-14).

[58] Jarek Nieplocha and Bryan Carpenter. "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems". In: *Parallel and Distributed Processing: 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing San Juan, Puerto Rico, USA, April 12–16, 1999 Proceedings*. Ed. by José Rolim, Frank Mueller, Albert Y. Zomaya, et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 533–546. ISBN: 978-3-540-48932-0. DOI: 10.1007/BFb0097937. URL: https://doi.org/10.1007/BFb0097937.

[59] Rajesh Nishtala, Hans Fugal, Steven Grimm, et al. "Scaling Memcache at Facebook". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[60] *Database Management System Choices Overview*. DBMS2, 2008. URL: http://www.dbms2.com/2008/02/15/database-management-system-choices-overview/ (visited on 2018-06-14).

[61] John Ousterhout, Arjun Gopalan, Ashish Gupta, et al. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015), 7:1–7:55. ISSN: 0734-2071. DOI: 10.1145/2806887. URL: http://doi.acm.org/10.1145/2806887.

[62] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, et al. "Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '16. Denver, CO, USA: USENIX Association, 2016, pp. 537–550. ISBN: 978-1-931971-30-0. URL: http://dl.acm.org/citation.cfm?id=3026959.3027008.

[63] Marius Poke and Torsten Hoefler. "DARE: High-Performance State Machine Replication on RDMA Networks". In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '15. Portland, Oregon, USA: ACM, 2015, pp. 107–118. ISBN: 978-1-4503-3550-8. DOI: 10.1145/2749246.2749267. URL: http://doi.acm.org/10.1145/2749246.2749267.

[64] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Analysis and Evolution of Journaling File Systems". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 8–8. URL: http://dl.acm.org/citation.cfm?id=1247360.1247368.

[65] *Pravega*. 2018. URL: http://pravega.io/ (visited on 2018-06-14).

[66] Dan Pritchett. "BASE: An Acid Alternative". In: *Queue* 6.3 (May 2008), pp. 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128. URL: http://doi.acm.org/10.1145/1394127.1394128.

[67] *Apache Pulsar*. Apache, 2018. URL: https://pulsar.incubator.apache.org/ (visited on 2018-06-26).

[68] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, et al. "Solving Big Data Challenges for Enterprise Application Performance Management". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1724–1735. ISSN: 2150-8097.

[69] *Redis*. 2018. URL: https://redis.io/ (visited on 2018-06-14).

[70] Ricci Robert and Eide Eric. "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications". In: *;login:* 39.6 (2014), pp. 36–38. ISSN: 1045-9219.

[71] *RDMA over Converged Ethernet*. 2018. URL: http://www.mellanox.com/page/products_dyn?product_family=79 (visited on 2018-06-14).

[72] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-structured File System". In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52. ISSN: 0734-2071. DOI: 10.1145/146941.146943. URL: http://doi.acm.org/10.1145/146941.146943.

[73] S.M. Rumble, D.F. Mazières, J.K. Ousterhout, et al. *Memory and Object Management in RAMCloud*. 2014.

[74] *Big Data - What Is It?* SAS, 2013. URL: http://www.sas.com/big-data/ (visited on 2018-06-14).

[75] P. Schulz, M. Matthe, H. Klessig, et al. "Latency Critical IoT Applications in 5G: Perspective on the Design of Radio Interface and Network Architecture". In: *IEEE Communications Magazine* 55.2 (2017), pp. 70–78. DOI: 10.1109/MCOM.2017.1600435CM.

[76] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, et al. "The Hadoop Distributed File System". In: *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 978-1-4244-7152-2. DOI: 10.1109/MSST.2010.5496972. URL: http://dx.doi.org/10.1109/MSST.2010.5496972.

[77] *The Square Kilometer Array Project*. 2018. URL: https://www.skatelescope.org/the-ska-project/ (visited on 2018-06-14).

[78] *SteamCharts: An ongoing analysis of Steam concurrent players*. Streaml, 2018. URL: http://steamcharts.com/ (visited on 2018-06-14).

[79] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. "The 8 Requirements of Real-time Stream Processing". In: *SIGMOD Rec.* 34.4 (Dec. 2005), pp. 42–47. ISSN: 0163-5808. DOI: 10.1145/1107499.1107504. URL: http://doi.acm.org/10.1145/1107499.1107504.

[80] Micheal Stonebraker. *What is Big Data*. Communications of the ACM, 2012. URL: https://cacm.acm.org/blogs/blog-cacm/155468-what-does-big-data-mean/fulltext (visited on 2018-06-14).

[81] *Messaging, storage, or both?* Streaml, 2017. URL: https://streaml.io/blog/messaging-storage-or-both/ (visited on 2018-06-14).

[82] Maomeng Su, Mingxing Zhang, Kang Chen, et al. "RFP: When RPC is Faster Than Server-Bypass with RDMA". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: ACM, 2017, pp. 1–15. ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064189. URL: http://doi.acm.org/10.1145/3064176.3064189.

[83] *The BDEC 'Pathways to Convergence'*. http://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/whitepapers/bdec2017pathways_v2.pdf.

[84] *The Environmental Toll of a Netflix Binge*. https://www.theatlantic.com/technology/archive/2015/12/there-are-no-clean-clouds/420744/.

[85] *The Evolution of Kafka at ING Bank*. https://www.confluent.io/kafka-summit-london18/the-evolution-of-kafka-at-ing-bank. (Visited on 2018-06-29).

[86] *The OpenCompute Project*. http://www.opencompute.org/.

[87] Eno Thereska, Austin Donnelly, and Dushyanth Narayanan. "Sierra: Practical power-proportionality for data center storage". In: *ACM European Conference on Computer Systems (EuroSys '11)*. 2011, pp. 169–182.

[88] Aniruddha N. Udipi, Naveen Muralimanohar, Niladrish Chatterjee, et al. "Rethinking DRAM Design and Organization for Energy-constrained Multi-cores". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10. Saint-Malo, France, 2010, pp. 175–186. ISBN: 978-1-4503-0053-7.

[89] Yandong Wang, Li Zhang, Jian Tan, et al. "HydraDB: A Resilient RDMA-driven Key-value Middleware for In-memory Cluster Computing". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '15. Austin, Texas: ACM, 2015, 22:1–22:11. ISBN: 978-1-4503-3723-6. DOI: 10.1145/2807591.2807614. URL: http://doi.acm.org/10.1145/2807591.2807614.

[90] *What Does It Cost to Build a Data Center?* http://www.sphomerun.com/data-center-sales-and-marketing-blog/what-does-it-cost-to-build-a-data-center.

[91] *When Streams Fail: Implementing a Resilient Apache Kafka Cluster at Goldman Sachs*. https://www.infoq.com/articles/resilient-kafka-goldman-sachs.

[92] Daniel Wong and Murali Annavaram. "KnightShift: Scaling the Energy Proportionality Wall Through Server-Level Heterogeneity". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Vancouver, B.C., CANADA, 2012, pp. 119–130. ISBN: 978-0-7695-4924-8.

[93] *Notes On Kafka Replication*. Wuciawe, 2018. URL: https://wuciawe.github.io/kafka/2017/03/09/notes-on-kafka-replication.html (visited on 2018-06-26).

[94] Tatu Ylönen. "Concurrent Shadow Paging: A New Direction for Database Research". In: (1992).

[95] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, et al. "Big Data: Astronomical or genomical?" In: *PLOS Biology* 13.7 (2015), pp. 1–11.

[96] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28. ISBN: 978-931971-92-8. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[97] Hao Zhang, Bogdan Marius Tudor, Gang Chen, et al. "Efficient In-memory Data Management: An Analysis". In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 833–836. ISSN: 2150-8097. DOI: 10.14778/2732951.2732956. URL: http://dx.doi.org/10.14778/2732951.2732956.

[98] Heng Zhang, Mingkai Dong, and Haibo Chen. "Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA, Feb. 2016. ISBN: 978-1-931971-28-7.

[99] Yibo Zhu, Haggai Eran, Daniel Firestone, et al. "Congestion Control for Large-Scale RDMA Deployments". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 523–536. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787484. URL: http://doi.acm.org/10.1145/2785956.2787484.

List of Publications

Papers in International Conferences

- **Yacine Taleb**, Ryan Stutsman, Gabriel Antoniu, and Toni Cortes. *Tailwind: Fast and Atomic RDMA-based Replication*. In Proceedings of the 2018 USENIX Annual Technical Conference, (**USENIX ATC'18**). July 2018, Boston, United States.
- **Yacine Taleb**, Shadi Ibrahim, Gabriel Antoniu, and Toni Cortes. *Characterizing Performance and Energy-efficiency of The RAMCloud Storage System*. In Proceedings of the The 37th IEEE International Conference on Distributed Computing Systems (**ICDCS'17**). May 2017, Atlanta, United States.

Workshops and Demos at International Conferences

- **Yacine Taleb**. *Optimizing Fault-Tolerance in In-memory Storage Systems*, in the EuroSys 2018 Doctoral Workshop (**EuroDW'18**). April 2018, Porto, Portugal.
- **Yacine Taleb**, Shadi Ibrahim, Gabriel Antoniu, Toni Cortes. *An Empirical Evaluation of How The Network Impacts The Performance and Energy Efficiency in RAMCloud*, Workshop on the Integration of Extreme Scale Computing and Big Data Management and Analytics in conjunction with IEEE/ACM **CCGRID'17**. May 2017, Madrid, Spain.
- **Y. Taleb**, S. Ibrahim, G. Antoniu., T. Cortes. *Understanding how the network impacts performance and energy-efficiency in the RAMCloud storage system*. Big Data Analytics: Challenges and Opportunities, held in conjuction with ACM/IEEE **SC'16**. November 2016, Salt Lake City, United States.