Schnelle parallele Fehlererholung in verteilten In-Memory Key-Value Systemen

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von
Kevin Beineke
geboren in
Düsseldorf

Düsseldorf, Juli 2018

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Berichterstatter:

1. Prof. Dr. Michael Schöttner

2. Prof. Dr. Martin Mauve

Tag der mündlichen Prüfung: 17. Dezember 2018

Abstract

Big data analytics and large-scale interactive graph applications require low-latency data access and high throughput for billions to trillions of mostly small data objects. Distributed in-memory systems address these challenges by storing all data objects in RAM and aggregating hundreds to thousands of servers, each providing 128 GB to 1024 GB RAM, in commodity clusters or in the cloud. This thesis addresses two main research challenges of large-scale distributed in-memory systems: (1) fast recovery of failed servers and (2) highly concurrent sending/receiving of network messages (small and large messages) with high throughput and low latency.

Masking server failures requires data replication. We decided to replicate data on remote disks and not in remote memory because RAM is too expensive and volatile, resulting in data losses in case of a data center power outage. For interactive applications, it is essential that server recovery is very fast, i.e., the objects' availability is restored within one or two seconds. This is challenging for servers storing hundreds of millions or even billions of small data objects. Additionally, the recovery performance depends on many factors like disk, memory and network bandwidth as well as processing power for reloading the storage.

This thesis proposes a novel backup and recovery concept based on replicating the data of one server to many backup servers which store replicas in logs on their local disks. This allows a fast-parallel recovery of a crashed server by aggregating resources of backup servers, each recovering a fraction of the failed server's objects. The global replica distribution is optimized to enable a fast-parallel recovery of a crashed server as well as providing additional options for tuning data loss probability in case of multiple simultaneous server failures.

We also propose a new two-level logging approach and efficient epoch-based version management both designed for storing replicas of large amounts of small data objects with a low memory footprint. Server failure detection, as well as recovery coordination, is based on a superpeer overlay network complemented by a fast, parallel local recovery utilizing multiple cores and mitigating I/O limitations. All proposed concepts have been implemented and integrated into the Java-based in-memory system DXRAM.

The evaluation shows that the proposed concept outperforms state-of-the-art distributed in-memory key-value stores. Large-scale experiments in the Microsoft Azure cloud show that servers storing hundreds of millions of small objects can be recovered in less than 2 seconds, even under heavy load.

The proposed crash-recovery architecture and the key-value store itself require a fast and highly concurrent network subsystem enabling many threads per server to synchronously and asynchronously send/receive small data objects, concurrently serialized into messages and aggregated transparently into large network packets. To the best of our knowledge, none of the available network systems in the Java world provide all these features.

This thesis proposes a network subsystem providing concurrent object serialization, synchronous and asynchronous messaging and automatic connection management. The modular design is able to support different transport implementations, currently implemented for Ethernet and InfiniBand. We combine several well-known and novel techniques, like lock-free programming, zero-copy sending/receiving, parallel de-/serialization and implicit thread scheduling, to allow

low-latency message passing while also providing high throughput.

The evaluation of the developed network subsystem shows good scalability with constant latencies and full saturation of the underlying interconnect, even in a worst-case scenario with an all-to-all communication pattern, tested with up to 64 servers in the cloud. The network subsystem achieves latencies of sub 10 µs (round-trip) including object de-/serialization and duplex throughputs of more than 10 GB/s with FDR InfiniBand and good performance with up to hundreds of threads sending/receiving in parallel, even with small messages ($<$ 100 bytes).

Zusammenfassung

Big-Data-Analysen und große interaktive Graphanwendungen erfordern Datenzugriffe mit niedriger Latenz und hohem Durchsatz für Milliarden bis Billionen von zumeist kleinen Datenobjekten. Verteilte In-Memory-Systeme bewältigen diese Herausforderung, indem sie alle Datenobjekte im Arbeitsspeicher halten und Hunderte bis Tausende von handelsüblichen Servern, mit jeweils 128 GB bis 1024 GB Arbeitsspeicher, in Clustern oder in der Cloud aggregieren. Diese Arbeit befasst sich mit zwei grundsätzliche Forschungsherausforderungen von großen verteilten In-Memory-Systemen: (1) schnelle Wiederherstellung ausgefallener Server und (2) nebenläufiges Senden/Empfangen von Netzwerknachrichten (kleine und große Nachrichten) mit hohem Durchsatz und geringer Latenz.

Das Maskieren von Serverausfällen erfordert Datenreplikation. In dem vorgeschlagenen Konzept werden Daten auf entfernten Festplatten bzw. SSDs und nicht im entfernten Speicher repliziert, da Arbeitsspeicher zu teuer und flüchtig ist, was zu Datenverlusten im Falle eines Stromausfalls im Rechenzentrum führt. Für interaktive Anwendungen ist es wichtig, dass die Serverwiederherstellung sehr schnell erfolgt, d. h. die Verfügbarkeit der Objekte innerhalb von ein bis zwei Sekunden wiederhergestellt wird. Dies ist eine Herausforderung für Server, die Hunderte von Millionen oder sogar Milliarden von kleinen Datenobjekten speichern. Zusätzlich hängt die Wiederherstellungsgeschwindigkeit von vielen Hardware-Faktoren wie Festplatten-, Speicher- und Netzwerkbandbreite sowie der Rechenleistung, welche für das Laden der Objekte in den Arbeitsspeicher benötigt wird, ab.

Diese Arbeit schlägt ein neuartiges Backup- und Wiederherstellungskonzept vor, das darauf basiert die Daten von einem Server auf viele Backup-Server zu replizieren, welche die Replikate in Logs auf ihren lokalen Festplatten speichern. Dies ermöglicht eine schnelle, parallele Wiederherstellung eines abgestürzten Servers durch Aggregation der Ressourcen von Backup-Servern, die jeweils einen Teil der Objekte des ausgefallenen Servers wiederherstellen. Die globale Backupverteilung ist optimiert, eine schnelle, parallele Wiederherstellung eines abgestürzten Servers zu ermöglichen und zusätzlich die Wahrscheinlichkeit eines Datenverlustes im Falle mehrerer gleichzeitiger Serverausfälle steuern zu können.

Diese Arbeit schlägt außerdem einen neuen zweistufigen Logging-Ansatz und ein effizientes Epochen-basiertes Versionsmanagement vor, die beide für die Speicherung von Replikaten vieler kleiner Datenobjekte mit geringem Speicherbedarf entwickelt wurden. Die Erkennung von Serverausfällen sowie die Koordination der Wiederherstellung basiert auf einem Superpeer-Overlay-Netz. Das Konzept wird ergänzt durch eine schnelle, parallele lokale Wiederherstellung, welche für Mehrkern-Prozessoren optimiert ist und durch die parallele Ausführung die Lesezeiten von Festplatte weitestgehend verdeckt. Alle vorgeschlagenen Konzepte wurden implementiert und in das Java-basierte In-Memory-System DXRAM integriert.

Die Auswertungen zeigen, dass die vorgeschlagenen Konzepte deren von modernen verteilten In-Memory Key-Value-Speichern überlegen ist. Große Experimente in der Microsoft Azure Cloud zeigen zudem, dass Server, die Hunderte von Millionen kleiner Objekte speichern, auch unter hoher Last in weniger als 2 Sekunden wiederhergestellt werden können.

Die vorgeschlagene Crash-Recovery-Architektur und der Key-Value-Speicher selbst erfordern ein schnelles und hoch-paralleles Netzwerk-Subsystem, mit dem viele Threads pro Server synchron

und asynchron kleine Datenobjekte gleichzeitig senden/empfangen können. Zusätzlich müssen die Datenobjekte parallel in Nachrichten serialisiert und transparent zu großen Netzwerk-Paketen aggregiert werden können, um hohe Durchsätze mit kleinen Objekten zu erzielen. Nach bestem Wissen bietet keines der für Java verfügbaren Netzwerksysteme alle diese Funktionen.

Diese Arbeit schlägt ein Netzwerk-Subsystem vor, das die parallele Serialisierung von Objekten, synchrones und asynchrones Nachrichtenversenden und automatisches Verbindungsmanagement bietet. Der modulare Aufbau unterstützt verschiedene Transport-Implementierungen, die derzeit für Ethernet und InfiniBand implementiert sind. Es werden verschiedene bekannte und neuartige Techniken, wie nicht-blockierende Synchronisierung, Senden/Empfangen ohne das Kopieren von Daten, parallele De-/Serialisierung und implizites Thread-Scheduling kombiniert, um das Senden/Empfangen von Nachrichten mit niedriger Latenz bei gleichzeitig hohem Durchsatz zu ermöglichen.

Die Auswertung des entwickelten Netzwerk-Subsystems zeigt eine gute Skalierbarkeit mit konstanten Latenzen und voller Sättigung des zugrundeliegenden Netzwerkes, selbst im schlimmsten Fall mit einem jeder-zu-jedem-Kommunikationsmuster, getestet mit bis zu 64 Servern in der Cloud. Das Netzwerk-Subsystem erreicht Latenzen von unter 10 µs (Umlaufzeit) inklusive Objekt-De-/Serialisierung und Duplex-Durchsatz von mehr als 10 GB/s mit FDR InfiniBand und gute Performance mit bis zu hunderten von Threads, die parallel senden und empfangen, auch bei kleinen Nachrichtengrößen (< 100 Bytes).

Danksagung

Ich möchte mich herzlichst bei Herrn Prof. Dr. Michael Schöttner für die Betreuung dieser Dissertation bedanken, für den persönlichen Einsatz und die konstruktive Unterstützung. Des Weiteren bedanke ich mich bei den Kollegen Stefan Nothaas und Florian Klein für die gute Zusammenarbeit. Alle besagten Personen hatten einen wesentlichen Anteil an dieser Arbeit und meiner persönlichen Entwicklung in dieser Zeit.

Weiterer Dank gilt Angela Rennwanz für Ihre organisatorische und Michael Braitmeier für die technische Unterstützung. Außerdem bin ich den zahlreichen Studenten für den gegenseitigen Wissenstransfer zu Dank verpflichtet.

Besondere Unterstützung habe ich von meiner Familie, insbesondere von meinen Eltern Marion und Jürgen Beineke, und meinen Freunden genossen. Herzlichsten Dank für den Rückhalt und die aufgebrachte Geduld!

Contents

Chapter 1.

Introduction

With the rise of web applications like search engines [89] and social media networks [116, 57] and the collection of continually increasing amounts of customer [69], health [50], spatial [131] and sensor data [86], big data analytics have become indispensable for many companies and introduced new challenges for the research community. Storing petabytes of data is a challenge itself, but big data analytics and large-scale interactive applications also require fast processing and low-latency data access to these huge amounts of data for real-time interactions [105]. The growing capacities and the falling prices of Dynamic Random Access Memory (DRAM; referred to as RAM throughout this thesis) allow loading large amounts of data from disk to RAM in order to improve data access times [128]. RAM is around two orders of magnitude faster than hard disk drives (HDD) and latencies are six orders of magnitude lower [46]. The difference is even higher for random access. In comparison to much faster Solid State Drives (SSD; also called flash memory or flash drive; the term disk is used for SSDs and HDDs), RAM is still one order of magnitude faster, and latency is 1000x lower [46]. Operators of social media networks like Facebook [116] and Twitter [57], as well as, other companies hosting large-scale web applications like Google [22] and Amazon [28] reached the limits of disk-based storages and started using in-memory caching-techniques [128, 80]. However, caches are difficult to synchronize with disk storages and the latencies between read and write accesses vary significantly, as well as, between read accesses served by the cache and cache misses [80]. Depending on the data access patterns, caches can grow very large to hold the cache miss ratio low. Facebook, for example, cached around 75% of all data on up to 1,000 Memcached [37] servers in 2009 [87].

Storing *all* data objects in RAM is a logical next step but is not a new idea as **in-memory** databases are a research subject since the 1980s [29, 32]. However, in-memory storages have become an important tool recently because of the large amounts of RAM available in commodity servers of private clusters and in the cloud. Various big data applications consist of billions or even trillions of mostly small objects [80, 75] rendering storing all objects in one server impossible. In data centers or in the cloud, servers are often connected with high-speed interconnects like 10 GBit/s Ethernet or 56 GBit/s InfiniBand [112], allowing low-latency access across the entire cluster. But, connecting hundreds of servers to store large-scale interactive applications or to execute big data analytics, increases the server failure probability significantly. This thesis focuses on two key aspects of **distributed** in-memory storages:

1. Fast parallel crash recovery: The primary goal is to mask server failures transparently within a few seconds to ensure the quality of service for the application. For instance, big data analytics benefit from the very low latency of the in-memory storage in a way that

single server failures do not noticeably affect the application if failures are handled within one to two seconds.

2. Fast and parallel network subsystem: Storing all data in RAM allows low-latency access on a single server. To enable low-latency access across the entire cluster, the distributed in-memory storage needs a fast and efficient network subsystem. This is essential for many graph algorithms and applications, e.g., social networks, because they have an irregular access pattern resulting in a lot of cross-traffic [116].

Both subjects are interdependent as remote replication and parallel recovery also cause a high load on the network and benefit from low latencies and high throughputs. Thus, it is important to match the concepts of both subjects.

1.0.1. Application Domains Processing Many Small Data Objects

Numerous concepts for replicating and recovering in-memory data [84, 61, 100] and low-latency interconnects [31, 87] have been proposed. Yet, solutions have a limited set of application domains in which they are efficient (discussed in Section 1.0.3 and 1.0.4). Therefore, in this Section, we introduce the primary application domains addressed within this thesis.

More and more big data applications demand low-latency access to mostly small objects [128], for instance, in the form of states (e.g., status information) or attributes (e.g., object metadata) in a social network [4]. These applications can be subdivided into two categories: online processing and offline analytics. Many applications can make the transition from offline analytics to online processing by using low-latency storages [88]. The first category primarily comprises user-generated data like in social media networks [116, 57], health applications [50] and social gaming [36]. A pleasant user experience requires keeping latencies for data accesses low [105]. Further online processing applications include real-time bidding [129], sensor data processing [86], online state management [102] and graph processing for information retrieval on a web graph and finding social influencers in a social media graph. The applications of the second category benefit from low-latency access by reducing the runtime of analytics considerably. Big data analytics can be found in various fields like advertising [69], telecommunications [102] and bio-informatics (e.g., enumerating common molecular substructures [33]). Commonly, big data applications have a graph-based data model with graph traversal algorithms (depth-first and breadth-first search), are based on data mining (e.g., PageRank [89], random walk, centrality measures, degree distribution, etc.) or machine learning (e.g., neural networks, deep learning, topic modeling, etc.) [123].

Graph applications based on social media, sensor and neural networks, typically, consist of many small data objects (the vertices of the graph) and even smaller edges. Some social media networks have billions of users and store trillions of objects [80]. In Facebook, a production workload analysis showed that around 70% of all requests served by the cache were less than 64 bytes in size, 99% less than 1 KB [80]. Bronson et al. [20] recorded 6.5 million requests from TAO [20], the geographical cache system of Facebook, showing that 45% of all edges of the social graph were empty and the rest of the edges had an average size of 97.8 bytes. Vertices were larger. Still, more than half of all vertices were smaller than 256 bytes. They also stated that 99.8% of all requests were reads. The analysis of Atikoglu et al. [4] covering 284 billion

requests confirms the small requests sizes, especially for user-account status information, and that read access dominates over write access. Another example is the web graph. In 2005, the web graph consisted of more than 11.5 billion indexed web pages which are highly connected [44]. Meusel et al. analyzed a web graph with 3.5 billion web pages and 128.7 billion links which fit entirely in one terabyte of RAM [75] showing the small vertex and edge sizes. The distributed in-memory storage needs to be able to efficiently manage small objects (< 100 bytes) in order to store graphs of this size.

1.0.2. Categorization of In-Memory Storages

Generally, in-memory storages can be categorized into two groups: in-memory databases and RAM-based NoSQL storages.

In-memory databases, also called data management systems, include relational databases (e.g., SAP HANA [107]), graph-based databases (e.g., GraphLab [64] and GraphX [125]), stream-based databases (e.g, GridGain [99]) and document-based databases (e.g., MongoDB [90]). Especially, relational in-memory databases share many design aspects with traditional databases despite the different primary memory [128]. Database management systems both disk-based and in-memory usually implement the ACID consistency model based on transactions offering atomicity, consistency, isolation and durability for all database operations. However, distributed databases require two-phase commits to apply transactions across multiple database instances, which limits the scalability considerably [92].

RAM-based NoSQL storages use hash tables, tables or key-value tuples as a data model [128]. Generally, other models are built upon the simple core data model (e.g., natural graphs). In contrary to database management systems, NoSQL storages usually implement the BASE consistency model providing basic availability, soft-state and eventual consistency favoring scalability over consistency [92]. This thesis focuses on but is not limited to distributed in-memory key-value stores which are a subset of this second category. The most related distributed in-memory key-value stores for this thesis are Redis [21], Aerospike [117] and RAMCloud [84] which are introduced in Section 1.0.3.

1.0.3. Fault-Tolerance Mechanisms

Distributed in-memory storages have specific characteristics which have to be considered when designing fault-tolerance mechanisms.

- Performance is critical: the main advantage of in-memory storages in comparison to traditional disk-based storages is the low latency and high throughput. Therefore, the replication overhead must be as low as possible to avoid impacting the performance considerably. Additionally, object availability must be restored as fast as possible after server failures.

- Scalability: some in-memory storages are able to distribute the load to thousands of servers to store billions to trillions of objects. The fault-tolerance mechanisms must scale

as well as other parts of the distributed storage.

- RAM is volatile and scarce: even with decreasing RAM prices, it is still more expensive than disks (HDDs and SSDs), and RAM capacities are much lower. A commodity server, today, has around 256 GB of RAM and several terabytes of disk storage [61]. Thus, occupying large amounts of RAM for backup mechanisms increases the number of servers to be aggregated or the cost of the servers by increasing the RAM size. In both cases, also the maintenance costs of running the servers are higher. Consequently, the underutilized disks should be used for storing replicas and backup metadata. Performance-sensible metadata, stored in RAM, should be minimized, especially if the application consists of billions of small data objects. Furthermore, when writing all data objects to disk, they are persistent and can be recovered after a power failure, even if all servers were affected.

In this thesis, we propose a backup and recovery scheme considering the key aspects performance, scalability and efficient RAM usage for applications with billions of small data objects accessed in write-heavy random or zipfian distributions [116, 75]. However, most of the concepts are versatile and can be implemented for various application domains and storage systems.

In the following, we introduce several existing techniques and in-memory storage systems with relevant fault-tolerance mechanisms.

Failure detection: many distributed in-memory storages use overlays and heartbeats in order to detect server failures. In this thesis, we cover consistent fail-stop, crash and omission failures. Byzantine and arbitrary failures are beyond the scope of this thesis but are well studied and can be prevented, for instance, by using a quorum and checksums [68]. Other failure sources like network failures and partitions are not addressed because they are very seldom in data centers [40] and solutions exist [130, 95].

Writing to disk must be optimized for the application pattern in order to provide high throughput. Especially with many small data objects, data aggregation and sequential write patterns are necessary to achieve optimal performance of disks favoring a log structure on disk. A log is a data structure to record events or objects by appending them at the end. Distributed in-memory storages typically use a circular log with a static size. When the end of the log is reached new objects are written to the beginning. A reorganization is executed periodically removing old and to be deleted objects from the log to compact it. A log is the preferred data structure for backups on disk because the sequential (and possibly buffered) writing allows high disk utilization.

With significant advances in flash memory over the past years, SSDs have become very attractive for backup purposes as the write throughput for sequential access is not far from RAM and network throughput [111]. Most in-memory storages use logs on SSDs, but the replica distribution and recovery mechanisms differ significantly.

We distinguish two crash recovery models:

1. Cold start: in-memory data is written to disk locally. After a single server failure, the server and the storage system is restarted. During the start of the storage system, all objects are restored by reading them from disk and loading them back to RAM. Reloading

all objects on a single server can be slow. The distributed in-memory key-value store Aerospike [117], for instance, takes more than 40 minutes to restore one billion objects [35]. Some systems can maintain exact copies of a server, which store the same objects in RAM and on disk, to improve availability. Obviously, storing all objects multiple times in RAM (and on disk) is very expensive. Furthermore, this approach might require operator assistance to restart the failed server and the storage.

2. Instant Recovery: the instant recovery model replicates the data of each server to remote servers' disks (ideally without storing them in RAM). In case of a server failure, multiple servers, each storing a part of the data on disk, recover the data in parallel. In this thesis, we propose backup and recovery concepts for many small data objects using the instant recovery model.

In the next paragraphs, we describe state-of-the-art in-memory storages which implement the cold start or instant recovery model.

SAP HANA [107] is an in-memory database which was designed for analytic and transactional access in a highly scalable environment and is widely used in the industry (e.g., Bayer, Dell, Reuters, Vodafone [101]). SAP HANA can distribute the load between multiple servers by using multiple index servers or by partitioning and distributing the database. SAP HANA offers a uniform data model based on row- or column-oriented tables and provides a temporal view by maintaining a history for tables. SAP HANA implements the cold start recovery model and uses a combination of checkpointing, delta backups which are incremental and/or differential backups and redo logs, all stored persistently on disk [100]. Checkpointing is a fault-tolerance technique saving snapshots of the current state in predefined scenarios (e.g., periodically). An incremental backup stores all changed data since the last incremental backup or checkpoint (for the first incremental backup) and is immutable whereas a differential backup contains all changed data since the last checkpoint and grows with every change. Several incremental backups can be combined to one differential backup. A redo log does not store data but changes to the data, written asynchronously to disk after every committed transaction. If a database instance fails, the server will be restarted and the data restored by loading the most recent checkpoint, all incremental and differential backups younger than the checkpoint and applying the changes recorded in redo log entries younger than the last delta backup [100].

Redis [21] is one of the most popular distributed in-memory systems which is used as an in-memory database, as a cache or publish-subscribe service by many companies like Twitter, GitHub, Stack Overflow and Flickr [124]. Redis provides data structures (numbers, strings, hash tables, lists and sets) upon the key-value model, basic transactions to bundle operations, which are executed atomically, and server-side scripting. In Redis, requests are served asynchronously by a single thread limiting the scalability on multi-core CPUs (using one instance per core has a high overhead). Redis differs from in-memory caches like Memcached [37] by providing on-disk persistence. The extension Redis Cluster provides data sharding based on a hash slot partition strategy requiring manual re-sharding for server upscaling [128]. Sharding describes the partitioning and distribution of large datasets to multiple servers. Redis implements the cold start recovery model. It provides a master-slave asynchronous replication and different on-disk persistence modes. To replicate in-memory objects, exact copies of masters, called slaves, are filled with all objects asynchronously. To overcome power outages and server failures, Redis provides snapshotting and append-only logging with periodical rewriting. However, to replicate on disk the server's objects must also be replicated in RAM which increases the total

amount of RAM needed drastically. Redis reads in a full log to compress it which is fast but introduces a lot of RAM overhead again. A crashed server is not recovered automatically but on restart. Further, the recovery is not able to recover one server in parallel on multiple slaves [15, 14].

Aerospike [117] is a distributed database platform providing consistency, reliability, self-management and high-performance clustering. Aerospike uses Paxos [59] consensus for server joining and failing, and balances load with the migration of partitions. Aerospike enables different storage modes for every namespace. For instance, all data can be stored on SSD with indexes in RAM or all data can be stored in RAM and optionally on SSD with a configurable replication factor [15]. Aerospike can be configured to log the in-memory objects to remote servers' disks [117] by creating and maintaining shadow copies of the servers. The logging architecture is optimized for flash memory. Like Redis, Aerospike does not offer a possibility to recover servers during ongoing operation but provides data restore on cold start on a single server [15, 14].

RAMCloud [87] is a distributed in-memory key-value store which uses a distributed hash table, maintained by a central coordinator, to map 64-bit object IDs to servers. RAMCloud implements a log-based replication of data on remote servers' disks [84] (instant recovery model). In contrast to Redis and Aerospike, RAMCloud organizes in-memory data also as a log which is scattered for replication purposes across many servers' disks in a master-slave coupling [98]. Scattering the state of one server's log on many backup servers allows fast recovery of large servers. However, the recovery is not optimized for small data objects. Thus, the recovery throughput decreases considerably for smaller objects [114]. Obviously, logging throughput depends on the I/O bandwidth of disks as well as on the available network bandwidth and CPU resources for data processing. RAMCloud uses a centralized log-reorganization approach executed on the in-memory log of the server which resends reorganized segments of the log over the network to backup servers. As a result, remaining valid objects will be re-replicated over the network after every reorganization iteration to clean-up the persistent logs on remote servers. This approach relieves remote disks but at the same time burdens the master and the network [15, 14].

FaRM [31] is a distributed in-memory computation platform by Microsoft Research. It combines the RAM of multiple servers to a shared address space. The memory of the participating servers is divided into 2 GB regions which are mapped using a form of consistent hashing with multiple virtual rings in a distributed hash table. FaRM uses slab, block and region allocators to support 256 distinct sizes from 64 bytes to 1 MB [31]. Therefore, the fragmentation for objects smaller than 64 bytes might be high. The data is accessed and modified by local and distributed transactions, implementing an optimized two-phase protocol, lock-free reads and shipping of transactions. FaRM uses replicated logging with strict serializability for transactions in order to replicate the data and the commit/redo log. FaRM claims to support instant parallel recovery of failed servers similar to RAMCloud, but no details have been published [31].

The Trinity Graph Engine [104] is a distributed in-memory graph database from Microsoft designed to support algorithms executed on graphs with billions of vertices. It uses a key-value data model implemented in C# on top of a memory cloud used for backup [104]. Every server stores a few equally sized trunks of the overall RAM. To find a key-value pair, Trinity uses two-level hashing [54]. Locally, the memory is organized as a log. Trinity uses a different approach by writing replicas into a memory cloud [104]. Thus, the complexity is transferred

to the memory cloud. On server failure, the leader machine reloads the data of the failed server by requesting the data from the memory cloud (instant recovery). Subsequently, the leader machine sends the reloaded objects to other servers and updates the metadata [104]. Apparently, the leader machine is the bottleneck in this approach (no parallel recovery).

Another different approach is implemented by Alluxio [61] which is a distributed file system. Alluxio provides fast data access times for all objects in cluster setups by holding all objects in RAM and avoiding replication to other servers and slower secondary storage through a lineage-based approach. This means that object updates are not stored on backups, but the operations applied to the object which are defined in job binaries like a MapReduce [27] or Spark [127] job. In case of a server failure, the data is reconstructed by re-executing the operations that generated the data, assuming the input data is immutable and the job deterministic. Additional asynchronous checkpointing to local disks limit the re-computation overhead. In case of a failure, the latest completed snapshot can be restored, and younger jobs be re-executed (instant recovery). While object creations and updates benefit from the replication-less approach, the recovery is impaired for high throughput scenarios as checkpointing falls behind (bound to I/O bandwidth). As a consequence, many objects have to be reconstructed based on possibly many jobs which have been executed since the last completed checkpoint. This reduces the recovery throughput, especially for many small objects [14]. Additionally, the lineage approach requires a job-based execution which limits the application domains.

1.0.4. Network Subsystems

Designing a network subsystem for in-memory storages is challenging as low-latency access to all objects is a key feature, as well as, providing high throughput. Logging and recovery even increase the challenge as data objects have to be replicated efficiently and quickly. Furthermore, in the primary application domains of this thesis, applications utilize many threads [128, 78, 108] and objects are small and, thus, messages often, too. This requires very efficient aggregation of messages, and the network subsystem has to cope with up to hundreds of threads. Interactive social media, large-scale graph applications, for instance, do not fit in a single server. Distributing the graph leads to a significant amount of cross-traffic as social media networks are rather dense [116]. Furthermore, the parallel execution of graph traversal algorithms like breadth-first search or random walk often result in an all-to-all communication pattern on graphs with a low diameter (e.g., the average distance between users on Facebook was just 4.7 hops in 2011 [116]). The objects (vertices and edges) of a social media graph are very small [20], so are the messages, especially when using a vertex-centric approach [67]. Another important aspect is the support for common data center network technologies like Ethernet and InfiniBand which benefits from a transport agnostic network layer. Further objectives are:

- Server address abstraction: using IDs/handles instead of network addresses
- Automatic connection management: create and open connections transparently on demand
- Scalability: low-latency access and high throughput must be provided for hundreds of servers, each running many threads, even in an all-to-all communication pattern
- Object serialization: efficiently serialize (complex) objects, relieving programs from reading

low-level network packets

- Application flow-control: avoid overburdening a server
- Low and predictable memory consumption

The concepts, proposed in this thesis, for a very fast message passing network subsystem are applicable to other network frameworks. Furthermore, the network subsystem is a stand-alone Java library and can be used for other Java applications or Java-based big data systems and is not limited to in-memory storages. We do not discuss the Message Passing Interface (MPI) or the message passing programming model here, but this can be found in Chapter 2.

In the following, we discuss the network subsystems of other in-memory storages (some of which are introduced in Section 1.0.3) and describe best practices.

Redis, Aerospike and Trinity use TCP sockets over Ethernet for network communication [41, 110, 42]. MICA [62] a single-server in-memory key-value store introduced several network stack optimizations to increase the performance of Ethernet networks. MICA bypasses sockets by accessing the NIC directly and reduces inter-core contention by using different core affinity techniques [62]. While this improves the performance, the applications need to implement transport layer features like retransmits

Wenhui et al. implemented a network communication module for Redis enabling RDMA over InfiniBand [115], but this module is not officially supported. FaRM also uses RDMA over InfiniBand and, additionally, supports RDMA over Ethernet with a Converged Ethernet (RoCE) link layer protocol. InfiniBand networks enable lower latencies and higher throughputs, e.g., 10 Gbit/s InfiniBand achieves 3.7 times more bandwidth and 5 to 6 times lower latencies than 10 Gbit/s Ethernet in simple ping-pong and throughput tests [26]. With RDMA, a server can directly access the memory of a remote server without involving the operating system or occupying CPU resources. Thus, RDMA accesses have very low latency and can achieve high throughputs. However, the use cases are limited because of the necessity to know the remote addresses which is hardly feasible for large clusters and billions or trillions of objects.

RAMCloud was specifically designed to utilize InfiniBand networks to serve remote procedure calls (RPC) in 5 to 10 µs [87] (RAMCloud also supports Ethernet networks). RAMCloud does not use RDMA for RPCs but uses message passing with messaging verbs. Many optimizations (e.g., kernel bypassing, lock-free sending/receiving, avoiding batches in favor of latency [87]) and a new network stack have been introduced to utilize an InfiniBand network efficiently.

While all introduced key-value stores are written in C, C++ or C#, many big data applications and platforms are written in Java [72, 122, 45] requiring communicating over a Java interface. Furthermore, the application needs to de-/serialize Java objects itself or use the included Java serializer which is slow and not space efficient [96]. In this thesis, we propose a Java network subsystem with a novel, fast and efficient serialization architecture optimized for concurrency.

1.0.5. Big Data Analytics in the Industry

Many companies executing big data analytics use the Hadoop ecosystem [72] (e.g., Amazon, eBay, Facebook, Google, IBM, LinkedIn, Microsoft, Twitter and Yahoo! [91]). It provides a great variety of frameworks for interactive computations [127], MapReduce [27, 30], machine learning [2, 74], searching/indexing [71, 43], streaming [23, 122] and graph [127, 125] applications on big data sets. The applications rely on SQL [3, 56] and NoSQL databases included in the Hadoop ecosystem like HBase [120] and Cassandra [58]. Apache Cassandra [58] is an open-source column-based NoSQL database which was developed by Facebook (but Facebook switched to HBase in 2010 [19]). Apache Spark [127] is a cluster computing framework which tries to overcome the limitations of the MapReduce paradigm by supporting interactive data analysis. The data is held in resilient distributed datasets (RDD), read-only sets of data objects scattered across the cluster and accessed in a restricted form of distributed shared memory [126]. Furthermore, the Hadoop ecosystem provides tools for scripting [38], scheduling [48], management and coordination [47, 121], as well as, for cluster resource management [118]. Most frameworks, databases and tools are based on the Apache Hadoop Distributed File System (HDFS) [106] which is a distributed file system designed to run on commodity hardware and applications with big data sets [63]. Typically, all data is stored on disks and depending on the application huge amounts of data are cached in RAM [37] [85] [66]. Memcached [37], for instance, is a distributed key-value cache used by many companies (Facebook [80], Twitter [73], Reddit [94], YouTube [103] etc.) in order to reduce latency. Another example is TAO [20] a geographically distributed cache designed and used by Facebook. Apache Spark can also hold RDDs in cache. Another approach is to replace HDFS by an in-memory file system like Apache Ignite [85] to employ a distributed in-memory storage.

Many big data analytics in the industry could benefit or already benefit from distributed in-memory storages and therefore could also profit from the proposed concepts regarding fast crash recovery and low-latency messaging.

1.1. Research Questions and Contributions

Big data analytics and large-scale interactive applications have high demands on the storage (see Section 1.0.1). Dr. Florian Klein proposed and designed the distributed in-memory key-value store DXRAM which stores small data objects with very low overhead and high performance [55, 53]. In this thesis, we present a logging and recovery concept for distributed in-memory key-value stores which have been integrated into DXRAM as a proof of concept. The proposed concepts are optimized for large amounts of small data objects and implement the instant recovery model allowing the recovery of crashed servers with 500,000,000 64-byte objects in less than two seconds (see Chapter 5). Additionally, we propose a versatile network subsystem with fast object de-/serialization optimized for small messages and support for high-speed networks (see Chapter 2). Naturally, the logging and recovery also benefit from an optimized network stack.

The management of billions of very small data objects opens new research questions in the two areas discussed in the following sections.

1.1.1. Backup and Recovery

In Section 1.0.3, we introduced different existing fault-tolerance mechanisms with various advantages and disadvantages. The cold start recovery model is easy to implement, but the reloading of all objects on the restarted server is too slow. Using shadow copies (occupying remote memory), on the other hand, requires too many resources for large-scale applications. Using a memory cloud for backups transfers the complexity from the key-value store to the memory cloud. Subsequently, directed replica placement in order to improve the recovery performance is not possible as the memory cloud handles replica placement. Furthermore, reloading all objects on one server and distributing the objects afterward is slow. The lineage approach allows very fast write accesses for many job-based application patterns as replicating the operation is generally faster than replicating the data. However, for given application domains the advantage is insignificant as objects are small and read accesses dominate over write accesses. On the other hand, the lineage approach impairs the recovery performance because the jobs have to be re-applied to the input objects which is CPU- and thus time-intense. We think RAMCloud's approach scattering the data objects of one server into remote logs on disk of many backup servers to be able to recover a server in parallel is a good foundation. Yet, many key aspects require to be adapted to DXRAM (discussed below) and primarily to the application domains opening many research questions. In the following, we discuss the most substantial research questions.

- **Backup zones:** How can we distribute billions of small objects to other servers for backup purposes? Determining backup servers for every single object is inefficient and requires storing the backup server tuples for each object which consumes a lot of memory. RAMCloud subdivides its in-memory log into 8 MB segments and replicates all objects of the entire segment to the same set of backup servers [98]. This approach is not applicable to DXRAM because the memory management differs significantly.

  Contributions: In this thesis, we propose a partitioning concept for storages with in-place memory management which subdivides the objects of one server into backup zones of a fixed size (e.g., 256 MB). This concept benefits from ascending object IDs by storing backup server affiliations in a B-tree bundling consecutive IDs in ranges in order to save memory. Every storage server manages its affiliations. However, for recovery initialization and coordination, a subset of this information has to be stored on another server. We contribute by proposing a failure detection and recovery coordination concept based on a virtual overlay.

- **Backup distribution:** The recovery performance depends considerably on the backup distribution as parallelism is crucial for the recovery performance. Large backup zones can be recovered more efficiently by many threads locally on a single server whereas small backup zones can be scattered to more backup servers to recover the data of a failed server in parallel. The backup distribution also has a substantial impact on the data loss probability, i.e., using fewer combinations of backup servers reduces the frequency of data loss events [24].

  Contributions: Both aspects are discussed in this thesis and the recovery performance is evaluated with varying numbers of backup servers.

- **Backup consistency:** Which consistency model should be applied? As RAM capacities are limited, storing in-memory replicas is infeasible. Reading replicas from disk to distribute the load onto multiple servers fails because of the higher latency and lower throughput of disks. Therefore, replicas are used for backup purpose, only. Nevertheless, a consistency model for distributing replicas has to be implemented. The strongest consistency is provided by linearizability which guarantees that all servers see all updates in the same order according to timestamps. However, the overhead of synchronizing clocks and ordering accesses is high [5]. Therefore, we implement sequential consistency which applies the same order of all updates for all servers. The difference is that the order can differ from the calling order of remote servers because it is inflicted by the server storing the in-memory object according to the FIFO principle.

- **Backup logs:** Is it feasible to store replicas on backup servers with a low memory overhead while having a high write throughput? The goal is to provide backup mechanisms that allow using backup servers also as storage servers storing its in-memory objects by occupying a small fraction of RAM and CPU time for backup purposes, only. We adopt the conventional approach of writing replicas to disks of backup servers by appending them to a log. However, storing all objects of possible many storage servers in one log on backup servers impairs the recovery performance because scanning the log requires reading it from disk entirely although a fraction of the log is recovered, only.

  Contributions: We describe a concept which sorts replicas by backup zones and stores every backup zone in a separate log. When using one log for every backup zone, one has to consider different access patterns for backup zones to ensure high throughput. We thoroughly evaluated different access patterns in order to optimize our approach (see Chapter 6).

- **Log reorganization:** How can we efficiently reorganize logs on backup servers? The reorganization of logs is important to avoid overfilling of a log and in order to keep recovery times low by removing invalid data frequently. RAMCloud reorganizes its in-memory logs and distributes reorganized segments to backup servers. This requires re-replicating of valid objects stressing the network.

  Contributions: We propose an orthogonal approach by reorganizing logs on backup servers. This reliefs the storage servers and the network, but increases the load on backup servers and the disk. Our approach has significant advantages during the recovery because backup servers have all the necessary information to recover and make available all valid objects of a backup zone without network communication. In RAMCloud segments with a set of arbitrary objects are recovered, i.e., all objects of the segment are read from disk and sent to a dedicated server which rejects invalid objects and distributes valid object to other servers to be loaded into the in-memory log. Furthermore, all recovered objects have to be replicated several times after the recovery whereas our concept replicates once, only. The evaluation shows that our approach is noticeably faster [14].

- **Log entry version control management:** How can invalid objects in logs of backup servers be identified? A log is subdivided into segments (e.g., 8 MB) to allow an incremental reorganization of a log which has a much lower memory overhead. Therefore, we cannot deduce the validity of an object based on the position within the log.

Contributions: We designed a version management which allocates ascending version numbers to incoming replicas to enable distinguishing valid from invalid object instances. The novel approach has a low memory footprint but also provides high throughput by using epochs in order to implement monotonic (not consecutive) version numbers.

1.1.2. Network Subsystem

Many distributed storages implement their own network subsystem. In this thesis, we describe a standalone network subsystem which can be used by any Java application and several concepts which can be adapted by different network subsystems. Conceptualizing a versatile network subsystem also applicable to the specific applications demands (see Section 1.0.1) opens several design and research questions:

- **Network transport:** Which network architectures should be supported? Most clusters are connected with Ethernet and/or InfiniBand [112]. Therefore, all introduced storages in Section 1.0.4 implement a transport implementation for one of these network architectures (RAMCloud supports both).

  Contributions: The network subsystem proposed in this thesis is designed modularly to support different network architectures. We currently support Ethernet and InfiniBand, and Loopback for evaluation purposes.

- **Communication model:** Different communication models have been proposed like message passing, RDMA or message passing over RDMA. Which communication models should be supported? CPU-driven message passing differs significantly from RDMA. Thus, a fraction of the network subsystem, only, can be shared when implementing both communication models. We decided to focus on traditional message passing because RDMA implicates several disadvantages when using it for an in-memory storage:

  – In large setups with billions or even trillions of objects, managing and gathering the remote addresses of all objects is bothersome (e.g., addresses are hard to combine to ranges, addresses of consecutively created objects cannot be determined directly) and requires a static memory layout. A static memory layout is not suitable for an in-memory storage because it obstructs resizing of objects and memory defragmentation.

  – Many operations of an in-memory storage cannot be reduced to remote memory access, i.e., message receipt often triggers an event which has to be executed by the CPU.

  – Often memory synchronization is required which is more complex (e.g., by using self-verifying data structures [76]) or slower (e.g., by acquiring a remote lock first) with RDMA.

  – The concept of message aggregation is not applicable to RDMA.

  – Many operations require more than one RDMA call. Typically, two RDMA calls are slower than one RPC [87].

  - RDMA over Ethernet requires compatible hardware. iWarp [93] and RoCE [119] is not widely supported in data centers.

  Some of the listed problems can be solved by implementing a message passing interface on top of RDMA (like FaRM [31]). However, this requires additional RDMA calls for sending a message to determine the memory address to write the message at negating the advantage of lower latency access.

- **Small messages:** How can we achieve high throughput and low latency even for small messages?

  Contributions: We propose a combination of known and novel techniques to improve throughput and latency for small messages:

  - Lock-free programming: the entire sending and receiving process of the proposed network system is a lock-free implementation. This increases the performance for multithreading applications and improves latency (e.g., checking an atomic value has a lower overhead than notifying a waiting thread).

  - Zero-copy: we propose a concept to de-/serialize messages directly into/from native memory which can be sent/received without copying.

  - Parallel de-/serialization: the serialization architecture proposed in this thesis allows efficient de-/serialization of messages with up to hundreds of threads. The de-/serialization process can be interrupted at any point and continue later in order to de-/serialize messages in smaller chunks sent/received over a congested network.

  - Message aggregation: messages are serialized into a ring buffer, one after another. This allows sending several messages in one large chunk of the ring buffer.

  - Implicit thread scheduling: lock-free operations can overburden the CPU caused by polling. We solve this with a situational multi-level waiting strategy.

  - Buffer and object pooling: to unburden the Java garbage collector, all data structures used for sending/receiving of messages are pooled or have a static size.

1.2. Structure of this Thesis

The focus of this thesis are the studied, developed and implemented crash recovery mechanisms of the in-memory key-value store DXRAM. DXRAM is introduced in Section 1.3 and is essential for all following chapters. The author of this thesis spent a lot of time and effort to improve the network submodule of DXRAM as it is one of the foundations of the distributed system and all proposed replication and recovery mechanisms highly depend on it. The network submodule is described in Chapter 2, the Ethernet transport implementation in Chapter 3. The replication of in-memory objects is detailed in the following chapter, followed by Chapter 5 which covers the recovery of failed servers. Both related publications ([15] and [14]) were written and the

described designs implemented and evaluated prior to the two network publications ([13] and [12]). Thus, not all discussed aspects of [13] and [12] affect the evaluations of [15] and [14]. Chapter 6 further discusses the replication and recovery but also brings together the revised network submodule and the backup by evaluating the concepts on high-speed I/O devices in [12]. Chapter 7 concludes this thesis and provides ideas for future work.

1.3. DXRAM

In this section, we present the distributed in-memory key-value store DXRAM [9] which is the foundation for the implementation of the concepts proposed in this thesis. The DXRAM project was started by Dr. Florian Klein in 2012 [52] and is supervised by Prof. Dr. Michael Schöttner. The author of this thesis, Kevin Beineke, joined the project during his master thesis in 2013 [7]. In this doctoral thesis, DXRAM was continuously developed, optimized and extended by Dr. Florian Klein (until June 2015), Stefan Nothaas (since October 2015) and Kevin Beineke. This section discusses the basic architecture of DXRAM and focuses on the relevant internals like the global metadata management, the memory management, the backup and the network subsystem, but also gives an overview of the DXRAM ecosystem. Concepts proposed in this thesis are included in this chapter. Some paragraphs are cited from project papers and are clearly labeled at the end of the paragraph.

DXRAM enables low-latency access to billions of small data objects on a commodity cluster or in the cloud by storing all objects in RAM, as key-value tuples. DXRAM can be used as a fast back-end key-value store or as an interactive compute platform. The latter allows coordinated, distributed and concurrent computations on storage servers and moving of compute tasks to reduce inter-server communications caused by excessive data transfer. Furthermore, DXGraph, an extended service of DXRAM, provides basic functionalities for loading and traversing a graph, stored naturally in the key-value store. Computations and graph processing can be combined by using a vertex-centric programming model coordinated by supersteps or a graph-centric model requiring fine-grained synchronization.

1.3.1. Overview

DXRAM is an open source distributed in-memory system with a layered architecture (see Figure 1.1), written in Java and available at GitHub [8]. Applications using DXRAM reside on top of the back-end storage or compute platform. Applications can access DXRAM services of the core or use extended services which are built on top of the DXRAM core. The extended services, as well as the DXRAM core, access submodules to implement one or multiple services which form the API for the applications.

Application Interfaces

DXRAM can be accessed in three different ways. Writing a DXRAM application which has access to all (extended) services (see Section 1.3.1), writing and executing a compute task with

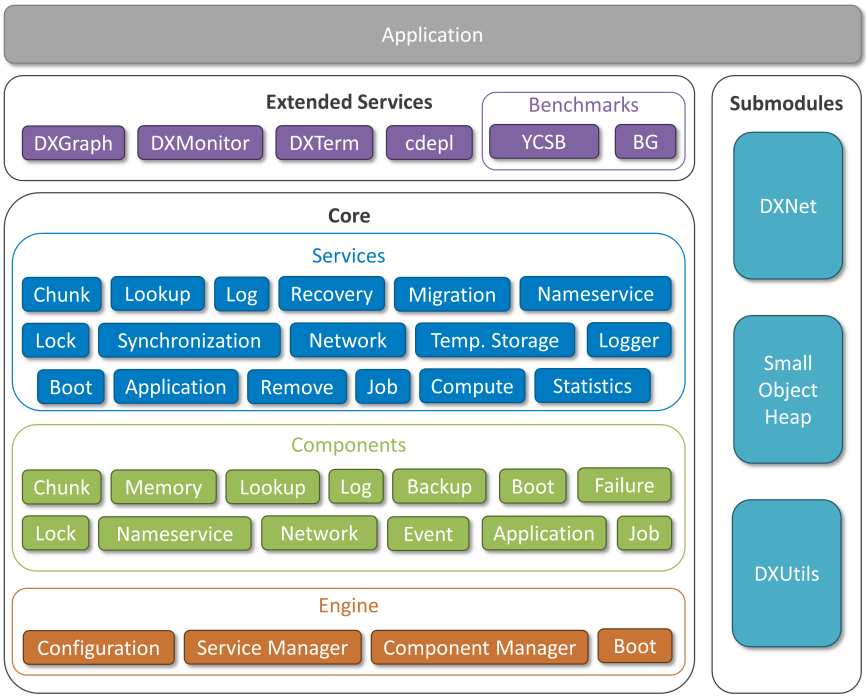

Figure 1.1.: DXRAM Architecture

explicit superstep synchronization (see Section 1.3.2) or accessing the system through a common line interface (CLI) which provides basic access to services.

Objects in DXRAM

In DXRAM, a key-value tuple is called *chunk*. A chunk consists of a 64-bit globally unique chunk ID and binary data. A chunk is the simplest data structure that can be stored in DXRAM. More complex data structures are provided, too, or can be defined by implementing de-/serialization methods (see Section 1.3.4). For example, for storing a graph, the data structures Vertex and Edge of DXGraph can be used.

Application

Applications using DXRAM are called DXApps, must implement the DXApp interface and are loaded during the runtime as separate compiled jar packages. A DXApp has access to the API of DXRAM, whereas the API is a collection of all methods the core services provide (see Figure 1.1). The API can be accessed by loading services as needed by the application. Furthermore, DXApps can also use the extended services. The application service of DXRAM loads DXRAM applications during startup of a storage server. One may run multiple and different applications on the same or different servers. DXApp instances run independently but can also be connected

and synchronized using services provided by the DXRAM core. For the configuration of a DXRAM application, DXRAM provides a configuration environment with JSON configuration files (see Section 8.1).

A DXRAM application can be a framework for other applications, as well. An example is the graph framework DXGraph. This application belongs to the extended services of DXRAM which are explained in the next section.

Extended Services

The extended services provide additional data models and functionalities beyond the key-value foundation of the DXRAM core.

DXGraph: DXGraph extends the compute service of DXRAM by adding data structures and algorithms for graph generation, loading and processing. Currently, it provides compute tasks for the compute service to load graph data from disk to DXRAM's key-value store and execute a multithreaded distributed breadth-first search (BFS) on a loaded graph. Vertices of the graph are represented naturally as vertex objects stored in DXRAM's key-value store [81]. The BFS algorithm is implemented as specified by the Graph500 benchmark [77] which allows us to use the BFS algorithm to compare DXGraph with other distributed in-memory graph frameworks like Grappa [79] and Graphlab [64]. The BFS algorithm is an informative benchmark because of its highly random access.

While the BFS is primarily a benchmark to evaluate the performance of DXRAM and DXGraph, the development of other graph applications is currently in progress, for example finding common molecular substructures for purposive drug designing [33]. This requires determining maximal cliques of two or more different graphs which is a common graph application and is NP-hard. Another graph application is the interpretation of the scientific context of higher-order citation graphs [109]. The scientific publishers and their relationships are represented in multi-dimensional tensor models, and scientific fields are identified with tensor decomposition methods. As a workload, the Microsoft Academic Graph with 120.887.883 papers from 119.892.201 unique authors is used. The result, semantically connected subgraphs, are then further analyzed.

The initial version of the DXGraph library including the BFS benchmark was designed by Stefan Nothaas. The application from bioinformatics emerges from cooperation with the workgroup of Prof. Dr. Gunnar W. Klau and Philipp Helo Rehs who joined the DXRAM team for his doctoral thesis. The second application was initiated by Prof. Dr. Dr. Sergej Sizov and is ported to DXRAM by Mikel Bahn.

Other Data Models: Other data models as the plain key-value tuples and natural graphs can easily be implemented on top of DXRAM. We provide a dynamic list data structure and an index is currently in development by Kai Neyenhuys. Tables and sets allocated to tablets and subsets will be implemented in the future.

Benchmark Interfaces: In order to compare DXRAM with other distributed in-memory key-value stores like RAMCloud [84] or Redis [21], we implemented DXRAM interfaces for the Yahoo! Cloud Serving Benchmark (YCSB) [25] and the BG benchmark [6].

The YCSB was designed to quantitatively compare distributed serving storage systems [25]. The benchmark offers a set of simple operations (reads, writes, range scans) and a tabular key-value data model to evaluate online storage systems regarding their elasticity, availability and replication. Furthermore, the YCSB is easily extensible for new storage systems and workloads [15].

The BG benchmark evaluates the performance of data storages for interactive social media networks [6]. It is developed at the Computer Science Department of the USC and supports a variety of state-of-the-art storage systems like MongoDB [90] and Cassandra [58]. Furthermore, BG has an open architecture and is extensible to be used with other storage systems. It follows an application-like approach by simulating interactive social networking actions known from, for example, Facebook, Twitter and YouTube. These include actions for viewing a profile, inviting a friend (or respectively following a user), posting a comment, etc. In order to evaluate a data store, it is loaded with a predefined number of users with optional profile pictures, friends and resources like holiday pictures or plain comments. A very large graph evolves by adding relationships between users or users and resources. How the graph is stored depends on the underlying storage system and is implemented by the specific storage client, e.g., with tables and indices for MongoDB and Cassandra or naturally with vertices and edges for DXRAM. After the load phase has finished, several BG clients start executing social networking actions to evaluate the data store. The BG coordinator increases the workload turn-based by adding BG clients (physical servers) and/or socialities (threads) until the data store is overburdened. Socialities are simulated users executing social networking actions. As the throughput of a data store does not give detailed insights about the distribution of response times, BG defines a Service Layer Agreement (SLA), which specifies the percentage of requests that have to be answered in a given time. For example, 95% of all responses have to arrive within 100 ms after sending the request. This way, BG can determine the maximal throughput that satisfies the predefined SLA, which represents a much fairer comparison value (called Social Action Rating) than the plain throughput. The distribution of social media actions can be configured as well (e.g., zipfian distribution) [53].

Additional extended services are listed and described in the Appendix in Section 8.1.

Core

DXRAM's core functionality is implemented as services and components. Services form the API for applications and can be dynamically enabled as needed. Services can communicate with services of the same type on remote servers but not with other service types (neither locally nor remotely) to avoid dependencies on the API. Components, on the other hand, can be accessed, locally, by all services and other components. Components implement local functionalities and can also be enabled on demand. All core services and components are listed in the Appendix in Sections 8.1 and 8.1.

Engine: The DXRAM engine provides the foundation to run components and services which implement the actual functionality for the DXRAM system. The engine bootstraps using a JSON formatted configuration file (see Section 8.1). A list of components and services including their specific configurations are loaded into the DXRAM context. This allows enabling/disabling of components or services to configure a DXRAM instance according to an application's

requirements. Configuration parameters for components or services are embedded within each class. After bootstrapping with a configuration file, DXRAM initializes all components using a fixed order which ensures resolving component dependencies correctly. Next, all services are initialized and the boot sequence is terminated by entering the main application loop in the DXRAM class.

Submodules

The submodules are used in DXRAM and can be used by DXApps and extended services of DXRAM. Additionally, the submodules are designed to be part of any Java application.

DXNet: DXNet is the network submodule of DXRAM and is described in Section 1.3.6.

Small Object Heap: The Small Object Heap is the major part of the efficient memory management for small objects which is described in Section 1.3.4.

DXUtils: DXUtils is collection of utility classes used in DXRAM, DXNet, extended services and applications of DXRAM. DXUtils contains classes for object serialization, accessing native memory through Java.unsafe [70], collecting statistics, unit conversion, a bloom filter and many more. Furthermore, it includes JNI interfaces for accessing a disk directly, bypassing the kernel's page cache (see Section 1.3.5), generating Cyclic Redundancy Check (CRC) checksums with SSE4.2 CPU instructions and for thread pinning and priority manipulation.

1.3.2. Compute Platform

DXRAM provides services allowing executing parallel computations on storage servers. The compute platform, which was developed by Stefan Nothaas, is composed of two services. The compute service allows distributing and coordinating computations to a set of servers. The job service deploys lightweight jobs locally. Naturally, the compute and job service can be used intertwiningly.

Compute Service

DXRAM's compute service is integrated into the DXRAM architecture adding functionalities to execute computations locally and also remotely on storage servers. If a computation involves more than one server, multiple servers have to be coordinated. The compute service, also called *master-slave service*, implements compute groups within the DXRAM network topology consisting of one coordinator (master) and an arbitrary number of compute servers (slaves). The master server controls the slave servers of its group by managing the joining/leaving of slaves to the compute group, accepting compute tasks, scheduling compute tasks to all slaves and synchronizing slaves between compute tasks. When writing a compute task, the programmer has access to the current compute group's unique ID, the slave ID assigned to the server as well as node IDs of all other slaves of the current compute group and all of the core DXRAM

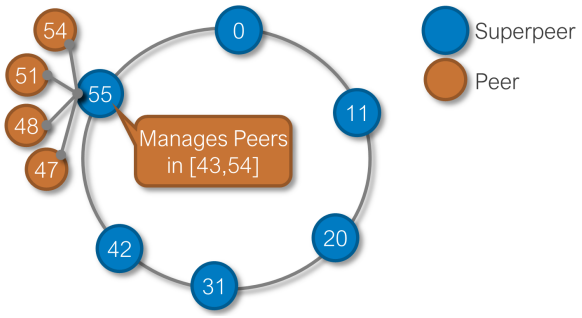

Figure 1.2.: Superpeer Overlay

services. The programmer can use the IDs as indices for partitioning his data or controlling the computation flow. Furthermore, slaves have access to all chunks, also from storage servers outside the compute group [81].

Compute tasks are submitted to compute groups and run concurrently on a single core on every slave server of the compute group. Naturally, a task is able to create threads to improve the CPU utilization. The compute service provides a superstep synchronization for all slaves of a compute group before and at the end of each task. Compute tasks are written in Java and are semantically coherent, e.g., loading of data, a processing step, a map phase, a reduce phase or printing of data, statistics and results. Multiple tasks can be combined in a JSON task script. Tasks can simply be executed sequentially or condition-based (e.g., execute the task if the return value of the previous task was > 0).

Job Service

The job service uses a per server configurable fixed size worker thread pool for deploying lightweight jobs. A job is implemented in Java and runs on a single server on a single core once per deployment. It also has access to all DXRAM services. A work-stealing approach implements local implicit load balancing between threads of the job service [60]. If a job needs to access data located on a remote server, the job can be delegated to the data-owning server. This improves data locality when executing the job and thus increases the performance [81].

1.3.3. Distributed Metadata Management

In the next sections (1.3.3 to 1.3.6), we discuss the most relevant aspects of the distributed key-value store DXRAM: the global metadata management, the local memory management, the backup and recovery mechanisms and the network subsystem.

This section is about the distributed metadata management which was originally designed by Dr. Florian Klein. The refinement of the concepts and the implementation was done by Kevin Beineke.

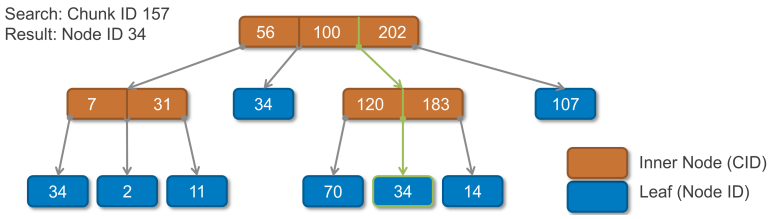

Figure 1.3.: Lookup Tree: Range-based (CIDs) lookup of Node IDs

In DXRAM, every server is either a *peer* or a *superpeer*. Peers store chunks, may run computations and exchange data directly with other peers, and also serve client requests when DXRAM is used as back-end storage. Peers can be *storage servers* (with in-memory chunks), *backup servers* (with logged chunks on disk) or both. Superpeers store global meta-data like the locations of chunks, implement a monitoring facility, detect failures and coordinate the recovery of failed peers, and also provide a naming service. The superpeers are arranged in a zero-hop overlay which is based on Chord [113] (see Figure 1.2) adapted to the conditions in a data center (e.g., every superpeer knows every other superpeers; low overhead to maintain as churn is seldom). Moreover, every peer is assigned to one superpeer which is responsible for meta-data management and recovery coordination of its associated peers. During server startup, every server receives a unique node ID [12].

Every superpeer replicates its data on a configurable number (default three) of succeeding superpeers in the overlay. If a superpeer fails, the first successor will automatically replace it and repair the overlay. In case of a power outage, the meta-data can be reconstructed based on the recovered peers' data. Thus, storing the meta-data on disk on superpeers is not necessary [12]. The metadata consists of location information, barriers for synchronization, a temporary storage (see Appendix 8.1) and a nameservice.

Locations Every chunk in DXRAM has a 64-bit globally unique chunk ID (CID). This ID consists of two separate parts: a 16-bit node ID of the chunk's creator and a 48-bit locally unique sequential number. With the creator's node ID being part of a CID, every chunk's initial location is known a-priori. But, the location of a chunk may change over time in case of load balancing decisions or when a server fails permanently. Superpeers use a modified B-tree (see Figure 1.3) [65], called lookup tree, allowing a space efficient and fast server lookup while supporting chunk migrations. Space efficiency is achieved by a per-server sequential ID generation and ID re-usage in case of chunk removals allowing to manage chunk locations using CID ranges with one entry for a set of chunks. In turn, a chunk location lookup will reply with a range of CIDs, not a single location, only. This reduces the number of location lookup requests. For caching of lookup locations on peers, a similar tree is used further reducing network load for lookups [12].

Barriers DXRAM provides barriers for DXRAM applications needing explicit synchronization, e.g., coordinating the start of a workload when all servers are initialized. The barriers are managed on the superpeers. Thus, every peer sends all requests to its superpeer. Each barrier is stored on the superpeer responsible for the peer who created the barrier and all barriers are

replicated to three succeeding superpeers to be able to continue if a superpeer crashes.

Nameservice In order to allow DXRAM applications to map a name on a chunk, the chunk ID is registered with the selected name in DXRAM's nameservice. The nameservice is managed by the superpeers, as well. It is optimized for the efficient handling of four byte IDs (integers) with a maximum of 2^{31}. Consequently, the usability with strings is limited to four bytes, too. To increase the number of characters to six, we permit digits, upper- and lower-case letters and "-", only. The nameservice must be configured either to store IDs or names before starting DXRAM. By limiting the names, we can reduce the overhead of the nameservice to a minimum without restraining most of the applications because they typically use the nameservice to register a few anchor points with static names or to register ascending or arbitrary IDs.

In many applications, all nameservice entries are registered by a few peers, only, e.g., by a coordinator. If all nameservice entries are stored on the responsible superpeers of those peers, the distribution would be inferior and overburden some superpeers. To register a chunk in DXRAM, the peer hashes the name/ID with CRC16 (16-bit Node ID) to identify the responsible superpeer. The superpeer adds the entry to a hash table and sends three replicas to its successors. The hash table uses linear probing and stores the name/ID (four bytes) and chunk ID (eight bytes) in three consecutive elements of an integer array to minimize the overhead.

In case all superpeers crash and the metadata has to be recovered, all nameservice entries are written to chunks which are automatically logged to disk. Every peer's first chunk, with chunk ID 0, contains all the peer's registered nameservice entries.

1.3.4. Efficient Memory Management

The memory management of DXRAM, which handles small data objects very efficiently, was initially designed by Dr. Florian Klein in his doctoral thesis [52]. As application demands changed over time, the memory management was extended and improved by Stefan Nothaas.

The sequential order of CIDs (as described in section 1.3.3) allows us to use compact paging-like address translation tables on servers with a constant lookup time complexity. Although, this table structure has similarities with well known operating systems' paging tables we apply it differently. On each DXRAM server, we use the lower part (LID) of the CID as a key to lookup the virtual memory address of the stored chunk data. The LID is split into multiple parts (e.g., four parts of 12 bit each) representing the distinct levels of the paging hierarchy. This allows us to allocate and free page tables on demand reducing the overall memory consumption of the local meta-data management. Complemented with an additional level indexed by node ID, storing of migrated chunks is possible as well. DXRAM uses a tailored memory allocator with very low footprint working on a large pre-reserved memory block outside the Java heap. For performance and space efficiency reasons, all memory operations are implemented using the Java Unsafe class [14]. The used and free blocks are connected by headers which contain information for the succeeding and preceding block to allow iteration in both directions (see Figure 1.4). The headers are very compact resulting in a metadata overhead (including CID tables) of around 5% for 64-byte chunks.

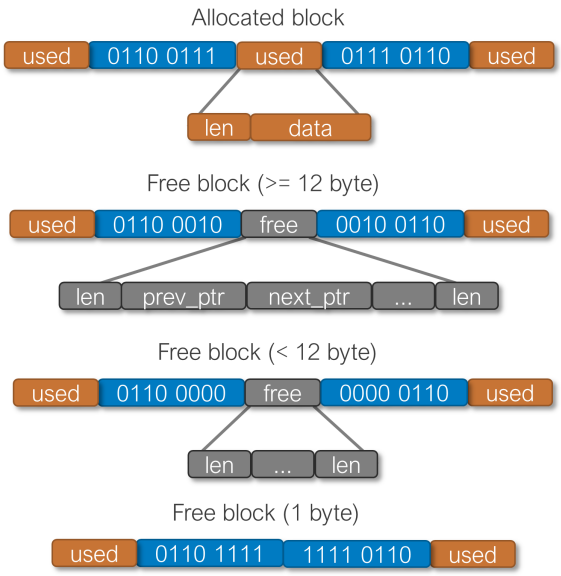

Figure 1.4.: Memory Structure

The local memory management is synchronized with a single read-write lock in the chunk service. The synchronization is necessary to avoid corrupting the metadata (CID tables and block management). Concurrent access to the same chunk needs to be handled by the application itself. Therefore, all read and write accesses are read-locked and create and remove accesses write-locked. Consequently, the read and write accesses are very fast, whereas creates and removes are slower especially when executed concurrently. Furthermore, creates and removes also slow-down reads and writes as they need exclusive access to the memory management. To alleviate the slow-downs, removes are executed asynchronously in a dedicated thread, i.e., every remove request is added to a queue and the dedicated thread processes the requests in bundles. Create operations cannot be executed asynchronously because consecutive read and write accesses would fail.

In order to increase the performance of create and remove operations, to enable fast and concurrent defragmentation and also to prepare DXRAM for Remote Direct Memory Access (RDMA), the synchronization of the memory management is being revised at the time of writing this thesis. In the future, concurrent access will be synchronized by read/write locking of single chunks in the CID table and locking of entire tables for metadata updates and batch processing. The locking will be based on Compare-and-Set (CAS) operations to reduce the overhead.

Data Structures

An object is stored as binary data in the in-memory heap of DXRAM and is referred to as chunk. A chunk consists of a unique chunk ID and binary data. The term chunk does not further describe the binary data itself. To store (complex) Java objects, we provide the data structure interface. Implementing the interface requires to define the de-/serialization of the

object from/to a byte array and the size of the serialized data. The serialization methods are called whenever the data structure is written to the in-memory heap or sent over the network (or written to a file etc.). Naturally, the deserialization methods are called for reading and receiving. We provide importer and exporter with de-/serialization methods for primitives, strings and arrays to simplify the definition of data structures. Typically, no reflection or type information is de-/serialized as the type is known from the context in most cases. However, if the type is unknown, e.g., when putting/getting arbitrary chunks in DXTerm, one has to add type information during the serialization.

At the time of writing this thesis, we are working on a unified declarative language for defining data structures and entire data models for DXRAM.

Migrations

Chunks can be migrated to another server for load balancing reasons or to improve access locality. DXRAM supports migrating single chunks, chunk ranges and the entire chunk set of a server. The migration can be initiated by an application or an extended service (e.g., DXTerm) or dynamically to balance the load based on DXMonitor's collected monitoring data.

Many fine-grained migrations might affect the performance and memory overhead of DXRAM's metadata management because the superpeers have to store all migrated ranges in the lookup trees. Furthermore, the client-side caches become invalid requiring an additional remote lookup. We expect the impact to be low, even for millions of migrations.

Locks

DXRAM provides a lock service which allows applications to lock single chunks. The chunks are locked locally on the owner server. To avoid affecting the performance during lock-free execution, lock operations are synchronized, only, i.e., a locked chunk can still be accessed with get/put operations. Locks are stored in a concurrent hash map. Thus, the lock service should not be used for an excessive amount of chunks as the hashmap has a relatively high overhead.

With the refined memory management, described above, the lock service can be implemented more efficiently, as well, by locking chunks directly in the CID table. However, this does not cover unlocking of chunks on server failure which will be addressed in the future.

1.3.5. Concurrent Backup and Recovery

The backup and recovery mechanisms of DXRAM were developed by Kevin Beineke and are the main contributions of this thesis. This section presents the basic ideas, only. Detailed descriptions can be found in Chapters 4, 5 and 6.

First, we describe the basic logging architecture of DXRAM which is subject of [15]. Below,

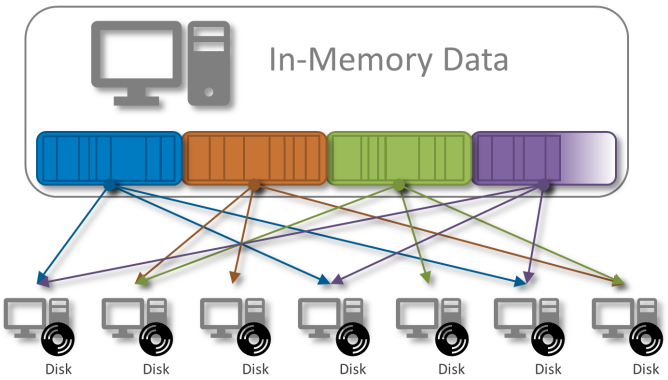

Figure 1.5.: The in-memory data of one server is subdivided into backup zones which are scattered to backup servers across the cluster.

we distinguish two different roles: *Masters* are storage servers which create and store chunks in RAM (see Section 1.3) and replicate them on *backup servers*. A backup server might also be a master and vice versa. In DXRAM, an in-memory data object is called a chunk whereas an object stored in a log on disk is referred to as log entry. The term disk is used for both Solid-State Drives (SSD) and Hard Disk Drives (HDD).

Replicating multi-billion small data objects in RAM is too expensive and does not allow to mask power outages. Therefore backup servers store backup zones on SSDs using logs to maximize write throughput.

Two-Level Logging

We divide every server's data into backup zones of equal size (see Figure 1.5). Backup zones are chunk collections which are stored in one separate log (one per backup zone) on every assigned backup server (default three per backup zone). Those logs are called *secondary logs* and are the final destination for every replica and the only data structure used to recover data from. By sorting backups per backup zone, we can speed up the recovery process by avoiding to analyze a single log with billions of entries mixed from several masters. The two-level log organization also ensures that infrequent written secondary logs do not thwart highly burdened secondary logs by writing small data to disk and thus utilizing the disk inefficiently. At the same time, incoming objects are quickly stored on disk to sustain power outages [12].

First, every object received for backup is written to a ring buffer, called write buffer, to bundle small request (Figure 1.6). This buffer is a lock-free ring-buffer which allows concurrently writing into the buffer while it is (partly) flushed to disk. During the flushing process, which is triggered periodically or if a threshold is reached, the content is sorted by backup zones to form larger piles of data in order to allow bulk writes to disk. If one of those piles is larger than a predefined threshold (e.g., 32 flash pages of the disk), it is written directly to the corresponding secondary log [12].

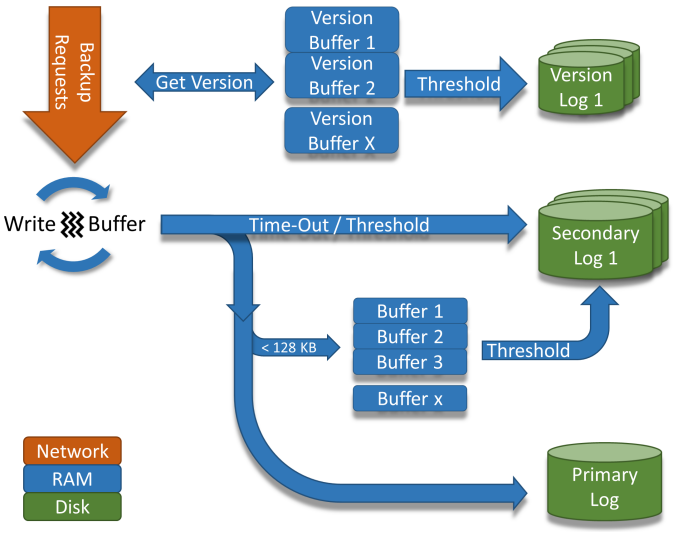

Figure 1.6.: Logging architecture. Every backup is buffered first. Depending on the amount of data per backup zone, the objects are either directly written to the specific secondary log or to primary log and to secondary log once there is enough data. Versions are determined by inquiring the corresponding version buffer, which is flushed to its version log frequently.

In addition to the secondary logs, there is one fixed-sized primary log for temporarily storing smaller piles of all backup zones to guarantee fast persistence without decreasing disk throughput. The smaller piles are also buffered in RAM separately, in so-called secondary log buffers, for every secondary log and will eventually be written to the corresponding secondary log when aggregated to a larger pile. Apparently, with this approach, some objects will be written to disk twice, but this is outweighed by utilizing the disk more efficiently. Waiting individually for every secondary log buffer until the threshold is reached without writing a copy to the primary log, is not an option as the data is prone to get lost in case of a power outage [12].

Backup-side Version Control

Masters do not store version information in RAM. As versions are necessary for identifying outdated data in the logs, the backup servers employ a version control used for the reorganization and recovery. A naïve solution would be to manage every object's version in RAM on backup servers. Unfortunately, this approach consumes too much memory, e.g., at least 12 bytes (8-byte CID and 4-byte version number) for every object stored in log easily summing up to many GB in RAM which is not affordable. Storing version information exclusively on disk, is also not practical because of performance reasons as this would require reads for each log write. Caching recent versions in memory could help for some access patterns but for the targeted application domain would either cause many read accesses for cache misses or occupy much memory.

We propose a novel version management concept running on every backup server and utilizing one version buffer per secondary log. The version buffer holds recent versions for this secondary

log in RAM until it is flushed to disk. In contrary to a simple cache solution, DXRAM's version management avoids loading missing entries from secondary storage by distinguishing time spans, called *epochs*, which serve as an extension of a plain version number. At the beginning of an epoch, the version buffer is empty. If a backup arrives within this epoch, its CID will be added to the corresponding version buffer with version number 0. Another backup for the same object within this epoch will increment the version number to 1, the next to 2 and so on. When the version buffer is flushed to disk, all version information is complemented by the current epoch, together creating a unique version. In the next epoch, the version buffer is empty again. This way, we create monotonic version numbers for consecutive write accesses to the same chunk. During the reorganization and recovery, all version numbers are read from disk to identify outdated log entries.

An epoch ends when the version buffer reaches a predefined threshold allowing to limit the buffer size, e.g., 1 MB per log. During flushing to disk, a version buffer is compacted resulting in a sequence of (CID, epoch, version)-tuples with no order. This sequence is appended to a file on disk, creating a log of unique versions for every single secondary log. We call it a version log. Over time, a version log contains several invalid entries which are tuples with outdated versions. To prevent a version log from continuously growing, it is compacted during the reorganization of the corresponding secondary log [12]. This new approach allows a high logging throughput by providing a fast version computation (without reading from disk) but saves a lot of memory in comparison to an in-memory version management (e.g., memory consumption is reduced by around 4000% for backup zones storing 64-byte objects). In contrary to a version management like done in RAMCloud, we can avoid tombstones with this approach. Tombstones are log entries without payload to mark an object as deleted in order to avoid recovering it [98]. A tombstone must not be deleted until all object versions are removed from the log which causes several problems (e.g., finding the tombstone when all object versions have been removed, deleting an object requires space in the log etc. [98]). The proposed approach does not need tombstones because deleted objects are marked in the version management which is written to disk persistently.

Recovery

As chunks are replicated to SSDs on remote servers, the recovery performance on a single server is limited by its hardware. Thus, like RAMCloud [84] and Google's Bigtable [22], DXRAM scatters the chunks from one server to many backup servers to aggregate SSD bandwidth and CPU processing power. Backup servers are not determined for each chunk but backup zones, containing up to 256 MB of chunks, to minimize meta-data overhead for backups. Hence, a server's data is split into 256 MB blocks which can be recovered from a backup server within 1 to 2 seconds. Many backup servers can perform this process in parallel allowing high recovery scalability. For every backup zone, three backup servers are assigned with a fixed replication and recovery order. Superpeers store the backup zones of each of their associated peers to avoid broadcasts during recovery. Thus, they can coordinate the recovery and directly contact the correct backups in case of a server failure. However, they do not need to store CIDs per backup zone; only storage servers need this information. Network limitations are masked by recovering a backup zone in the memory of backup servers and resume normal operation. Chunks can be migrated asynchronously to a fresh server later [14].

Every write access to a chunk (create, delete and update) is replicated to the backup servers of the particular backup zone, according to the replication order. To efficiently resolve the backup zone affiliation for billions of locally stored chunks, which is necessary to send the replicas to backup servers, every DXRAM server utilizes a B-tree which is optimized for storing CID ranges. This backup zone tree provides fast access times while being very space efficient because of (CID-)range aggregation (e.g., an entire backup zone with millions of chunks is stored with 1 to 2 entries within the B-tree) [14].

Server failures are detected and recovery is coordinated by the superpeer next in the superpeer overlay. This superpeer informs the responsible backup servers storing relevant backup zones. Then, the backup servers recover all valid chunks from the associated secondary logs into their local memory. All required information to initialize the recovery of a failed server is available a-priori on the superpeer as backup servers of all backup zones are stored on superpeers (including backups) as well. Thus, there is no need to gather information from backup servers (in contrary to RAMCloud and Google's Bigtable) [14].

The local recovery on a backup server is also challenging as regular secondary logs store several millions of small log entries and for every single log entry, the validity (currentness and status) and correctness (data integrity) have to be verified. To limit the temporary memory consumption, a secondary log is recovered segment by segment. The segments are processed by iterating over all log entries and restoring the valid log entries. The validity of a chunk is verified by reading all current version numbers from SSD (stored in a version log) before the recovery process and comparing them with the log entry version numbers. Obviously, gathering, storing, reading and comparing millions of version numbers is time critical. Furthermore, parallelization is crucial to speed up the recovery process and increase the overall system's performance and responsiveness by improving the availability of chunks [14].

Finally, the lookup meta-data of all recovered chunks must be updated on corresponding superpeers. The necessary network transfer can be minimized by, again, aggregating CIDs into ranges [14].

1.3.6. DXNet: Lock-free Messaging

The first version of the network subsystem of DXRAM was created by Dr. Florian Klein and Mark Ewert. Kevin Beineke and Stefan Nothaas redesigned and optimized the network subsystem to enable low-latency and high-throughput messaging. A detailed breakdown of the contributions and more detailed explanations can be found in 2 and 3.

DXNet is a network library implemented in Java and designed for DXRAM but can be used for other big data frameworks, too. DXNet implements an event-driven message passing approach and provides a simple and easy to use application interface. It is optimized for highly multi-threaded sending and receiving of small messages by using lock-free data structures, fast concurrent serialization, zero copy and zero allocation. Split into two major parts, the core of DXNet provides automatic connection management, serialization of message objects and an interface for implementing different transports. Currently, an Ethernet transport using Java.nio sockets and an InfiniBand transport using ibverbs is available [18].

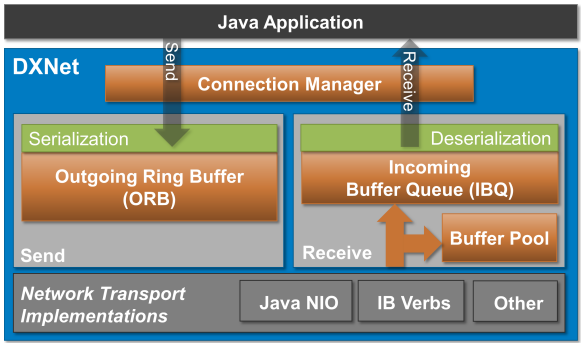

Figure 1.7.: Simplified DXNet Architecture (from [13])

This section describes the most important aspects of DXNet and its core (see Figure 1.7) which are relevant to this thesis. More details have been published in [13] and the source code is available at Github [11].

Automatic Connection Management

All nodes are addressed using an abstract 16-bit node ID. Address mappings must be registered to allow associating the node IDs of each remote node with a corresponding implementation dependent endpoint (e.g., socket, queue pair). To provide scalability with up to hundreds of simultaneous connections, our event-driven system does not create one thread per connection. A new connection is created automatically once the first message is either sent or received to/from a new destination. Connections are closed once a configurable connection limit is reached based on a recently used strategy. Faulty connections (e.g., remote node not reachable anymore) are handled and cleaned up by the connection manager. Error handling on connection errors or timeouts is propagated to the application using exceptions [18].

Sending of Messages

Messages are Java objects and sent asynchronously. A message can be targeted towards one or multiple receivers. Using the message type *Request*, it is sent to one receiver. The sender waits until receiving a corresponding response message (transparently handled by DXNet) or skips waiting and collects the response later [18].

One or multiple application threads can concurrently call DXNet to send messages. Every message is automatically and concurrently serialized into the Outgoing Ring Buffer (ORB), a natively allocated and lock-free ring buffer. When used concurrently, messages are automatically aggregated to increase send throughput. The ORB, one per connection, is allocated in native memory to allow direct and zero-copy access by the low-level transport. The transport runs one decoupled dedicated thread (one for all connections) which removes the serialized and ready to send data from the ORB and forwards it to the hardware [18].

Receiving of Messages

The network transport handles incoming data by writing it to pooled native buffers. We use native buffers and pooling to avoid burdening the Java garbage collection. Depending on how a transport writes and reads data, the buffer might contain fully serialized messages or just fragments. Every buffer is pushed to the Incoming Buffer Queue (IBQ) which is based on a ring buffer. Both, the buffer pool as well as the IBQ are shared among all connections. Dedicated message handler threads (configurable) pull buffers from the IBQ and process them asynchronously by de-serializing them and creating Java message objects. Finally, the messages are passed to pre-registered callback methods of the application [18].

Flow Control

DXNet implements its own flow control (FC) mechanism to avoid flooding a remote node with messages. This would result in an increased overall latency and lower throughput if the remote node cannot keep up with processing incoming messages. When sending messages, the per-connection dedicated FC checks if a configurable threshold is exceeded. This threshold describes the number of bytes sent by the current node but not fully processed by the receiving node. The receiving node counts the number of bytes received and sends a confirmation back to the source node in regular intervals. Once the sender receives this confirmation, the number of bytes sent but not processed is reduced. If an application send thread was previously blocked due to exceeding this threshold, it can now continue with processing the message [18].

Transport Interface

DXNet provides a transport interface allowing implementations of different transport types. One of the implemented transports can be selected at the start of DXNet. The transport and its specific semantics are transparent to the applications [18]. The transport API is described in the Appendix in Section 8.1.

Transport Implementations

DXNet has an open architecture supporting different network transport technologies. Currently, we have transport implementations for TCP/IP over Ethernet (using Java.nio), reliable verbs over Infiniband (based on JNI), and Loopback (for evaluation). We will only sketch some important aspects of these transports. More details of the Ethernet transport can be found in Chapter 3.

The Ethernet transport (EthDXNet) implementation is based on Java.nio and maps Direct-ByteBuffers to the ORB allowing to send data without copying it in userspace. Furthermore, two channels are opened for every connection to avoid channel duplication and for providing a side-band flow control channel for each connection. Channel duplication may occur when

two servers create connections to each other simultaneously. The second channel allows exchanging flow control messages necessary to maximize throughput on a connection by using the back-channel.

The InfiniBand transport accesses the IBDXNet library (C++) using JNI. IBDXNet utilizes ibverbs to implement direct communication using the InfiniBand HCA. IBDXNet uses one dedicated send and one dedicated receive thread, both processing outgoing/incoming data in native memory. Context switching from C++ to Java was designed carefully and is highly optimized to avoid latency.

The Loopback transport is used for the experiments allowing to study the performance of DXNet without any bottlenecks from a real network. Data is not sent over a network device nor the operating system's loopback device (latency would be considerably high) but is directly copied from the ORB to a pooled incoming buffer. Furthermore, the Loopback transport simulates a server sending and receiving messages at the highest possible throughput allowing to evaluate DXNet's performance [18].

Chapter 2.

Efficient Messaging for Java Applications running in Data Centers

This chapter summarizes the contributions and includes a copy of our paper [13].

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Efficient Messaging for Java Applications running in Data Centers". In: Cluster, Cloud and Grid Computing (CC-GRID), 2018 18th IEEE/ACM International Symposium on, Workshop AHPAMA 18. May 2018, pp. 589-598

2.1. Paper Summary

In this publication, we describe DXNet, a low-latency and high-throughput messaging system for Java applications. DXNet provides fast object de-/serialization, automatic connection management with node ID abstraction and asynchronous sending/receiving of messages as well as synchronous requests/responses. The API is small and intuitive enabling easy integration of DXNet into existing Java applications. For elastic applications, DXNet allows dynamic adding of servers and error propagation for lost connections.

The high performance of DXNet is achieved by using pooled data structures outside of the Java heap and an interruptible de-/serialization both enabling zero-copy sending/receiving, efficient work-sharing and highly parallel processing with lock-free endpoints and an adaptive thread scheduling to utilize the CPU cores efficiently. The evaluation shows message processing times of sub 300 ns and an aggregated throughput of more than 16 GByte/s for Loopback communication on a typical server instance. With 56 GBit/s InfiniBand, the throughput reaches 10.4 GByte/s and an average RTT of sub 10 µs (sub 5 µs one way). DXNet saturates Ethernet networks with >=256 byte messages and InfiniBand with >=2 KB, demonstrating the efficient processing of small messages.

2.2. Importance and Impact on Thesis

DXRAM's primary application domains are very demanding regarding the underlying network system (see Section 1.0.1). Information retrieval on large-scale graphs, for example, generates a significant amount of cross-traffic and latency must be low for the interactive requests. Furthermore, high throughput is necessary, if many requests have to be served in parallel. The demands are even higher, if objects are replicated to remote servers in order to provide fault-tolerance (see Chapter 4 to 5). While the object sizes and thus the message sizes are small for given applications, DXRAM's migration and recovery automatically aggregate objects before sending. We designed DXNet to address given requirements which are the support for different network technologies common in data centers like Ethernet and InfiniBand, high throughput and low latency for small and large messages and requests, a dynamic connection management for up- and downscaling, the support for multithreading (up to hundreds of threads) and a very efficient de-/serialization of Java objects. Certainly, many other Java applications would benefit from DXNet as well.

DXRAM's logging and recovery, which are the major contributions of this thesis, are highly depended on DXNet. Still, the evaluations in the papers [15] and [14] are based on an older network module as DXNet was developed afterward. Because of the rather slow I/O devices in the test environments (1 to 5 GB/s Ethernet and SSDs with less than 350 MB/s write throughput), the benefits in using DXNet as described in this publication would be small. In [12], we show the logging performance of DXRAM when using a PCI-e SSD and DXNet with a high-speed network.

2.3. Personal Contribution

DXRAM's initial network submodule was designed and implemented by Dr. Florian Klein in collaboration with Marc Ewert who dedicated his bachelor thesis [34] to the parallelization of the network submodule. The design was limited to Ethernet networks and optimized for Gigabit Ethernet. Furthermore, the network submodule was deeply integrated into DXRAM. The architecture and implementation were revised and replaced by Kevin Beineke, the author of this thesis, and Stefan Nothaas. The first step was replacing slow data structures, optimizing thread handling and buffer processing to improve throughput and latency. This was done primarily by Kevin Beineke, as well as, the second step which enabled failure detection and handling used for DXRAM's node recovery, for instance. In the third step, Stefan Nothaas refactored the network submodule to be (1) stand-alone and (2) modular regarding the underlying network technology (particularly InfiniBand). The last step was to redesign most of the data structures and the serialization, improve thread work-sharing and to reduce the overhead of inter-thread communication in order to saturate high-speed networks like >10 GBit/s Ethernet and >56 GBit/s InfiniBand. This publication focuses primarily on the last step.

Stefan Nothaas develops IBDXNet, DXNet's network transport for InfiniBand networks, which is not part of the publication other than being used for the evaluation. Therefore, Stefan Nothaas and Kevin Beineke worked in close collaboration to improve DXNet to be one of the fastest messaging systems for Java applications. This paragraph describes the apportionment

of work for this publication to the best of the knowledge of both mentioned authors. Stefan Nothaas initiated many optimizations by designing IBDXNet and discovering bottlenecks in the messaging system DXNet which were hardly detectable with slower networks. Furthermore, Stefan Nothaas introduced the idea of using lock-free data structures to improve the performance of DXNet. Despite contributing performance optimizations and debugging, Stefan Nothaas also implemented interfaces for the serialization and accessing direct ByteBuffers with Java.unsafe. Additionally, Stefan Nothaas invested much time in structuring the code, designing an error-detecting configuration of DXNet and a statistics module which is a useful tool to investigate DXNet's performance. Kevin Beineke designed and implemented the concurrent de-/serialization including work-flow optimizations and thread communication, the data structure pooling, the loopback and Java.nio transports and the basic DXNet benchmark. Furthermore, most of the ORB and CUB were designed by Kevin Beineke, with contributions by Stefan Nothaas. Other lock-free data structures were inspired by the ORB and implemented by both authors as well as the parking strategy which emerged in an incremental process.

Prof. Dr. Michael Schöttner took part in many discussions about the design and evaluation of DXNet.

Kevin Beineke structured and wrote most of the paper, including all figures but figures 2 and 3 whose initial design was contributed by Prof. Dr. Michael Schöttner. Prof. Dr. Michael Schöttner also helped in improving comprehensibility and reviewed the paper several times. Stefan Nothaas contributed in writing the initial versions of section I and II and designing a figure of DXNet's architecture which was used for figure 1 and 5. Stefan Nothaas reviewed the paper several times as well and helped improve it.

Efficient Messaging for Java Applications running in Data Centers

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—Big data and large-scale Java applications often aggregate the resources of many servers. Low-latency and high-throughput network communication is important, if the applications have to process many concurrent interactive queries. We designed DXNet to address these challenges providing fast object de-/serialization, automatic connection management and zero-copy messaging. The latter includes sending of asynchronous messages as well as synchronous requests/responses and an event-driven message receiving approach. DXNet is optimized for small messages (< 64 bytes) in order to support highly interactive web applications, e.g., graph-based information retrieval, but works well with larger messages (e.g., 8 MB) as well. DXNet is available as standalone component on Github and its modular design is open for different transports currently supporting Ethernet and InfiniBand. The evaluation with micro benchmarks and YCSB using Ethernet and InfiniBand shows request-response latencies sub 10 µs (round-trip) including object de-/serialization, as well as a maximum throughput of more than 9 GByte/s.**

***Keywords*—Message passing; Ethernet networks; InfiniBand; Java; Data centers; Cloud computing;**

I. Introduction

Today, many interactive applications are built upon very large graphs, e.g., social networks [1], comparing molecular structures in bioinformatics [2] or mobile network state management systems [3]. These graphs consist of billions of small data objects which are typically held in memory to provide low latency access. But, as data volumes grow fast it becomes necessary to aggregate many servers or move to expensive super computers. Usually, big data applications are executed in cloud data centers or on high performance clusters which provide fast networking with 10 GBit/s and beyond. Distributed and parallel processing of in memory data based on very fast networks requires the software stack to be designed carefully, especially if latency is important.

Many big data applications are written in Java and benefit from platform independence and a rich selection of libraries supporting the programmer in designing distributed and parallel applications [1], [4]–[8]. This includes many possibilities to exchange data between Java servers, ranging from high-level Remote Method Invocation (RMI) [9] to low-level byte stream processing using Java sockets [10] or the Message Passing Interface (MPI) for HPC applications [11]. DXNet does not aim at replacing any of those solutions but to rather complement the spectrum.

DXNet is a network library for Java-based applications which has originally been designed for DXRAM a distributed in-memory key-value store and DXGraph a graph processing framework built on top of DXRAM. We provide DXNet as a standalone library through GitHub [12] as we think it could be useful for many other Java-based big data applications.

The contributions of this paper are:

- the DXNet architecture (highly concurrent and transport agnostic);
- zero-copy, parallel de-/serialization of Java objects;
- lock-free, event-driven message handling;
- evaluations with 5 GBit/s Ethernet (Microsoft Azure) and 56 GBit/s Infiniband networks.

The evaluation shows that DXNet efficiently handles high loads with dozens of application threads concurrently sending and receiving messages. Synchronous request/response patterns can be processed in sub 10 µs RTT (Round-Trip Time) with Infiniband transport (including object de-/serialization). And, high throughput is achieved even with smaller payloads, e.g., bandwidth saturation with 1-2 KB payload on InfiniBand and 256 byte payload on Ethernet.

The structure of the paper is as follows: after discussing related work, we present an overview of DXNet in Section III. In Section IV, we describe the lock-free Outgoing Ring Buffer followed by the concurrent serialization in Section V. The next section explains the event-driven processing of incoming data. Sections VII and VIII present thread parking strategies and transport implementation aspects. Evaluation results are discussed in Section IX, followed by the conclusions.

II. Related Work

DXNet combines high-level thread and connection management and a concurrent object de-/serialization with lock-free, event-driven message handling and zero-copy data transfer over Ethernet and InfiniBand (extensible). To the best of our knowledge, no other Java-based network library provides these communication semantics. Because of space constraints, we compare DXNet with the most relevant related work, only.

Distributed shared memory (DSM) is re-gaining attraction due to networks supporting **RDMA** but is not an option for most existing Java applications as DSM requires a different application architecture and an integration in the heap management of the Java Virtual Machine (JVM) [13].

Java's RMI [9] provides a high level mechanism to transparently invoke methods of objects on a remote machine, similar to Remote Procedure Calls (RPC). Parameters are automatically de-/serialized and references result in a serialization of the object itself and all reachable objects (transitive closure) which can be costly [14]. Missing classes can be loaded from

remote servers during RMI calls which is very flexible but introduces even more complexity and overhead. The built-in serialization is known to be slow and not very space efficient [14], [15]. Furthermore, method calls are always blocking.

Manta [16] improves runtime costs of RMI by using a native static compiler. **KaRMI** [17], a drop-in replacement for Java RMI, is implemented in Java without any native code supporting standard Ethernet. KaRMI also replaces Java's built-in serialization reducing overhead and improving overall performance. DXNet does not provide transparent remote method calls but an efficient parallel serialization which avoids copying memory. DXNet is primarily designed for parallel applications and high concurrency, RMI for Web applications and services.

MPI is the state-of-the-art message passing standard for parallel high performance computing and provides very efficient message passing for primitive, derived, vector and indexed data types [18]. As MPI's official support is limited to C, C++ and Fortran, Java object serialization is not provided. Nevertheless, MPI is available for Java applications through implementations of the MPI standard in Java [19] or wrappers of a native library [20].

MPI-2 introduced multi-threading for MPI processes [18] enabling well-known advantages of threads. Prior to MPI-2, intra-node parallelization demanded the execution of multiple MPI processes (and the use of more expensive IPC). To enable multi-threading, the process has to call `MPI_init_thread` (instead of `MPI_init`) and to define the level of thread support ranging from single-threaded execution over funneled and serialized multi-threading to complete multi-threaded execution (every thread may call MPI methods at any time). A lot of effort has been put into the last mode to provide a high concurrent performance [21], [22]. Still, the performance is limited compared to a message passing service designed for multi-threading [21].

One of DXNet's main application domains are ongoing applications with dynamic node addition and removal (not limited to), e.g., distributed key-value stores or graph storages. The MPI standard defines the required functionality for adding and removing processes (over Berkeley Sockets with `MPI_Comm_join` or by calling `MPI_Open_port` and `MPI_Comm_accept` on the server and `MPI_Comm_connect` on the client). Unfortunately, most recent MPI implementations are still not supporting these features entirely [23], [24]. Furthermore, job shutdown and crash handling is also limited [24]. MPI is particularly suitable for spawning jobs with finite runtime in a static environment. DXNet, on the other hand, was designed for up- and downscaling and handling node failures. In [25], DXNet was used in the in-memory key-value store DXRAM to examine crash behavior and scalability.

High level mechanisms for typical **socket-like interfaces** supporting Gigabit Ethernet (and higher) are provided by Java.nio [26], [27], Java Fast Sockets (JFS) [28] or High Performance Java Sockets [29]. DXNet uses Java.nio to implement a transport for commonly used Ethernet networks.

III. Overview

DXNet relieves programmers from connection management, provides transferring Java objects (beyond plain Java.nio stream sockets) and allows the integration of different underlying network transports, currently supporting reliable verbs over InfiniBand and TCP/IP over Ethernet. In this section, we give a brief overview of the interfaces and functionality of DXNet (see Fig. 1). Further details can be found in the GitHub repository [12].

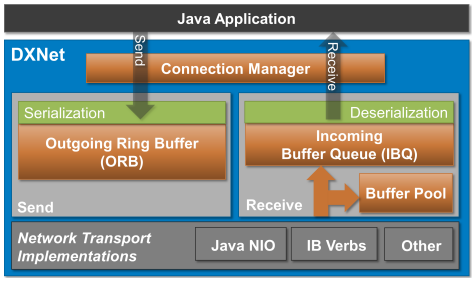

Figure 1. Simplified DXNet Architecture

A. Basic Functionality

Automatic connection management. DXNet abstracts physical network addresses, e.g., IP/Port for Ethernet or GUID for InfiniBand, by using nodeIDs. The aforementioned node address mappings are registered in the library and are mutable for server up- and downscaling. A new connection is opened automatically when a message needs to be sent to another server which is not connected thus far. In case of errors, the library will throw exceptions to be handled by the application. Connections are closed based on a recently used strategy, if the configurable connection limit is exceeded, or in case of network errors which may be reported by the transport layer or detected using timeouts, e.g., absent responses.

Sending messages. DXNet sends messages asynchronously to one or multiple receivers but also provides blocking requests (to one receiver) which return when the corresponding response is received (association of response and requests is transparently managed by DXNet). **Messages are Java objects and serialized** by using DXNet's fast and concurrent serialization (providing default implementations for most commonly used objects, see Section V). The serialization writes directly into the Outgoing Ring Buffer (ORB) which aggregates messages for high throughput (see Section IV) and is allocated outside of the Java heap. Sending data is performed by a decoupled transport thread based on event signaling. DXNet also includes a flow control mechanism, which is not further described here.

Receiving messages. When incoming data is detected by the network transport, it requests a pooled native memory buffer (avoiding to burden the Java garbage collector) and copies the data into the buffer (see Section VI and Fig. 1). The buffer containing the received data is then pushed to the Incoming Buffer Queue (IBQ), a ring buffer storing references on buffers which are ready to be deserialized (see Section VI). The buffer pool and the IBQ are shared among all connections. The buffers of the IBQ are pulled and processed asynchronously by dedicated threads. Message processing includes parsing message headers, creating the message objects and deserializing the payload data. Finally, the **received message is passed back to the application (as a Java object)** using a pre-registered callback method.

A brief overview of DXNet's API is shown in Table I.

TABLE I. DXNET'S APPLICATION INTERFACE

`new DXNet(config,nodeMap)`	initialize/configure (max. connections, server address mappings etc.)

`MyMessage extends Message/Request/Response`
`  exportObject(exporter)`
`  importObject(importer)`
`  sizeOfObject()`
	define message (serializable Java object) by implementing three methods
	serialize message with predefined methods from exporter
	deserialize message with predefined methods from importer
	return payload length

`sendMessage(message)`	send message asynchronously (receivers defined in message instance)
`sendSync(request,timeout)`	send request/response synchronously
`MyReceiver implements MessageReceiver` `  onIncomingMessage(message)`	receive messages/requests as Java objects pre-registered callback handler function

B. High Throughput and Low Latency

A key objective of DXNet is to provide high throughput and low latency messaging even for small messages found in many graph applications, for instance. We achieve this with a thread-based and event-driven architecture using lock-free synchronization, zero-copy, and zero-allocation.

Multithreading. All processing steps like serialization, deserialization, message transfer and processing are handled by multiple threads which are decoupled through events allowing high parallelism.

Lock-free event signaling. Dispatching processing events between threads is implemented using lock-free synchronization allowing low-latency signaling. CPU load is managed without impairing latency by parking currently idling threads.

Fast serialization. DXNet implements fast serialization of complex data structures and writes data directly into an ORB. The ORB can be accessed by many threads in parallel and ORBs are not shared between different connections increasing concurrency even more. The processing of incoming messages is also highly scalable because of the event-driven architecture.

Zero copy. DXNet does not copy data for messaging (except de-/serialization). For TCP/IP, we rely on Java's DirectByteBuffers and for InfiniBand on verbs pinning the buffers used by DXNet.

Zero allocation. DXNet uses object pooling wherever possible avoiding time-consuming instance creation and, even more important, not burdening the Java garbage collector which may block an application in case of low memory for multiple seconds.

C. Network Transport Interface

DXNet supports different underlying reliable network transports. The integration of a new transport protocol requires implementing just five methods:

- signal data availability on connection (callback);
- pull data from ORB and send it;
- push received data to IBQ;
- setup a connection;
- close a connection.

IV. LOCK-FREE OUTGOING RING BUFFER

The Outgoing Ring Buffer (ORB) is a key component for outgoing messages and essential for providing high throughput and low latency. The latter is achieved by a highly concurrent approach based on lock-free synchronization.

Each connection has one dedicated ORB allowing concurrent processing of different connections. The ORB itself allows many application threads serializing their outgoing objects concurrently and directly into the ORB. The ORBs are allocated outside of the Java heap in native memory allowing zero-copy sending by the network transport. Directly serializing Java objects into the ORB is more efficient than serializing each object in a separate buffer and combining them later by copying these buffers. The ORB preserves message ordering as given by the application threads and aggregates smaller messages in order to achieve high throughput. We decided to use lock-free synchronization for concurrency control which is more complex but more efficient with respect to latency compared to locks.

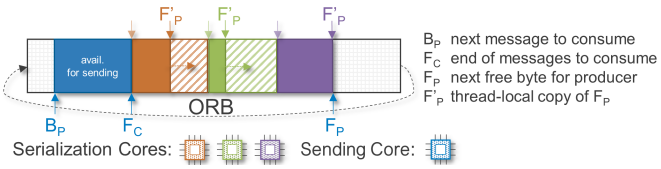

Figure 2. ORB for parallel serialization and aggregating outgoing messages.

A. Basic Lock-Free Approach

The ORB has a configurable but fixed size and is accessed concurrently by several producers (application threads) and one consumer (dedicated transport thread for sending messages). The configurable buffer size limits the maximum number of messages/bytes to be aggregated. For our experiments (see Section IX), we used 1 MB and 4 MB ORBs.

Fig. 2 shows the ORB with three application threads producing data (serialization cores). All pointers move forward from left to right with a wrap around at the end. The white area between F_P and B_P is free memory.

Messages available for sending (fully serialized) are found by the consumer (sending core) between B_P and F_C. The consumer sends aggregated messages and moves B_P forward accordingly but not beyond F_C. All messages between F_C and F_P are not yet ready for sending as parallel serialization is still in progress.

F_P is moved forward concurrently (if the buffer has enough space left) by the producers using a Compare-and-Set (CAS) operation, available in Java through `Unsafe` (see Section IV-C). Therewith, each producer can concurrently and safely store the position of F_P in a local variable F'_P and adjust F_P by its message size. All F'_P pointers (thread-local variables) are used by the associated producer for writing its serialization data concurrently at the correct position in the ORB. The light-colored arrows in Fig. 2 show the starting point of each serialization core (producer) whereas the solid-colored

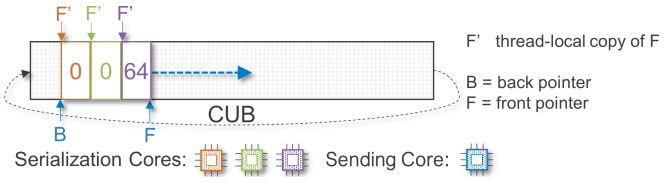

Figure 3. Catch-Up Buffer (CUB). Allowing faster producers returning early and not wasting CPU cycles for waiting.

ones show the current position. In the example, the purple producer finished its serialization first and the green and orange producers are still working.

F_C is moved forward by producers when messages are fully serialized. In Fig. 2, the purple producer finishes before the orange and green ones but cannot set F_C to F_P because the two preceding messages (from the other producers) have not been completely serialized yet. Each producer can easily detect unfinished preceding messages by comparing its starting point (light-colored arrow) with F_C. Obviously, the purple starting point is not equal to F_C. A naive solution lets fast producers wait for slower ones. As we do not want to impact latency we cannot use locks/conditions here. An alternative solution is to busy-poll until all preceding messages have been serialized. Finally, F_C can be set forward and the thread can return.

B. Optimized Lock-Free Solution

The basic solution already avoids the overhead of locks, but with increasing number of parallel serializations the probability of threads having to wait for slower ones increases. The busy-polling can easily overload the CPU. Reducing the polling frequency of producers by sleeping (> 1 ms) or parking (> 10 µs) increases latency too much. Instead, we propose a solution which avoids having fast producers waiting for slower ones by leaving a notice and returning early to the application. This notice includes the message size so that slower producers can move F_P forward for the faster ones. But, message ordering must be preserved.

Our solution is based on another configurable fixed-size ring buffer called Catch-Up Buffer (CUB). As mentioned before, we allocate one ORB for each connection which is now complemented by one associated CUB (e.g., with 1000 entries) for every ORB. The CUB is implemented using an integer array, each entry for one potential left-back notice from faster producers. An entry will be 0 if there is no notice or > 0 representing the message size if a producer finished faster than its predecessors. In the latter case, a slower producer will move forward F_P by the left-back message size.

Fig. 3 shows a CUB corresponding to the ORB shown in Fig. 2. The front pointer F is moved concurrently using a CAS operation (similar to F_P in the ORB). The colored F' are the thread-local copies needed by the producers to leave back a potential notice at the correct position in the CUB. The 64 is a notice from the purple producer (its message size, filled purple box in Fig. 2.) who finished fastest and returned already to the application. The green and orange producers are still working (0 = no notice). If the green producer would now finish before the orange one it would also fill in its message size and return immediately.

If the orange producer finishes next, it moves forward F_C in the ORB as well as B in the CUB (leaving no notice). The green one will do the same, but twice as it will detect the notice (64) after committing its serialization and thus move forward F_C in the ORB by 64 bytes and also B (now pointing to F in the CUB, indicating we are done).

It is important that the order of entries in the ORB and the CUB is consistent, meaning, we need to move forward F and F_P, as well as B and F_C synchronously. We do this, by storing each of those two indexes in one 64-bit long variable in Java and, as the CAS operation is working atomically on 64-bit longs, we can avoid locks.

Two more challenges remain, namely large messages which cannot be serialized at once and a potential ORB overflow during the serialization (both discussed in Section V).

C. Native Memory

The ORB is allocated in native memory (off Java heap) allowing the underlying network transports to send messages without copying them. The class `Unsafe` provides basic methods for memory allocation, memory copy and reading/writing primitives from/into native memory. Furthermore, `Unsafe` is very fast because of extensive optimizations and is widely used in third-party libraries [30].

We favor Unsafe over DirectByteBuffers [27] for two reasons. First, access is faster (e.g., missing boundary checks we already handle on higher level). Second, Unsafe is more versatile because it allows accessing memory which was allocated in C/C++ code (e.g., used for InfiniBand).

V. Serialization

DXNet is designed to send and receive Java objects which need to be de-/serialized from/into a byte stream of messages. The built-in serialization of Java (interface `Serializable`) as well as file-based solutions are too slow and have a large memory footprint [31] (because of automatic un-/marshaling and the use of separators). Other binary serializer like Kyro [32], for instance, either do not support writing directly into native memory or interruptible processing which is needed by DXNet (see Section V-A and V-B). We propose a new serializer addressing all these limitations while still being intuitive to use. The programmer has to implement two interfaces `Importable` and `Exportable`. The former requires implementing the method `importObject()`, the latter `exportObject()` and both `sizeOfObject()`.

A. Export

Exporter. The serialization (or export) of Java objects requires an `exporter` which is passed to `exportObject()`. The exporter class provides default method implementations for the serialization of all primitives, *compact numbers* and Strings and can be extended for supporting custom types (all types can also be arranged in arrays). Compact numbers are coded integers using a variable number of bytes as needed to reduce space overhead.

The exporter writes directly into the ORB by using Unsafe (see Section IV). It stores the start position within the ORB, the size of the ORB and the current position within the message.

Exporting an object involves two steps: exporting the message header (see Fig. 4) which has a fixed size and exporting the variable-sized payload by calling `exportObject()`.

DXNet uses its `default exporter` for serialization which is optimized for performance. It is complemented by

Figure 4. Message header. Cat.: message, request or response; X: exclusive or not (ordering).

two other exporters (described below) for handling messages which do not fit in the ORB without copying buffers.

Buffer overflow. If the end of the ORB will be reached during the serialization of an object, DXNet switches to the `overflow exporter`. The overflow exporter performs a boundary check for each data item of an object and writes bytes with a wrap-around to the beginning of the ORB, if necessary. The resulting message is sent as two pieces over the network stream avoiding copying data.

Large messages. Serialized objects resulting in messages larger than the ORB must be written iteratively. First, the entire unused section of the ORB (see Fig. 2) is reserved and filled with the first part of the message. If the back pointer is reached, the export is interrupted and its current state is stored in an `unfinished operation` instance to allow resuming serialization as soon as there is free space in the ORB again.

Unfinished operation. The instance stores the interrupt position within the message and the rest of the current operation. Depending on the operation, the rest is either a part of a primitive which can be stored in a `long` within the unfinished operation or an object with partly uninitialized fields whose reference can be stored.

Resume serialization after an interrupt. To continue the serialization, the method `exportObject()` is called again (threads return after being interrupted during serialization) and all previously successfully executed export operations are automatically skipped until the position stored in the unfinished operation is reached. The rest of the object is serialized from there (might be interrupted, again). For exporting large messages, the `large message exporter` is used, which extends the overflow exporter.

B. Import

All incoming messages are written into native memory buffers taken from the incoming buffer pool and are pushed to the IBQ (see Section VI). Each buffer contains received bytes (one or several messages) from the connection stream. The underlying network independently splits and aggregates packets resulting in a buffer beginning and ending at any byte within a message. DXNet is able to serialize split messages without copying buffers.

The import works analogously to the export. Messages are deserialized directly from native memory by using Unsafe (message header and payload). The fast `default importer` is complemented by three other importers (described below) for handling split messages. This requires to handle three situations: buffer overflow (tail of message/header missing), buffer underflow (head of a message/header is missing) and both combined.

Buffer overflow. When the buffer's end will be reached before the message is complete, we switch to the `overflow importer`. It does boundary checks and uses the unfinished operation (see Section V-A) when necessary. Furthermore, the serialization is aborted with an `IndexOutOfBoundsException` handled by DXNet

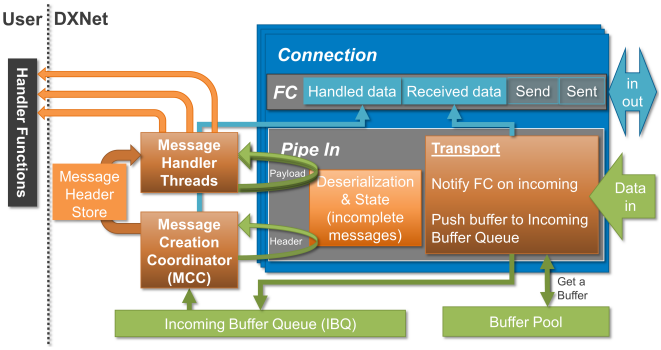

Figure 5. Receiving and processing messages. Green: Native memory access.

avoiding returning invalid values for succeeding operations.

Buffer underflow. This situation occurs after a buffer overflow (on the same stream). It is known apriori and handled by the `underflow importer`, which uses the unfinished operation instance (passed from the overflow importer) containing all information necessary to continue deserialization.

Buffer under- and overflow. When a message's head and tail is missing (likely for large messages), the message is handled by the `underoverflow importer`.

C. Resumable Import and Export Methods

Messages may be split caused by DXNet's buffering or the underlying network. In order to avoid copying buffers, we require both import and export methods to be interruptible and idempotent as they may be called multiple times for one object (to avoid blocking threads, see Sections V-A and V-B). DXNet's importer and exporter methods are sufficient for most object types, but custom object structures must be aware of this and avoid functions causing side effects (e.g., I/O access).

VI. Event-Driven Processing of Incoming Data

Fig. 5 gives an overview of the parallel event-driven processing of incoming data. Like for the ORB, we use multi-threading, lock-free synchronization, zero-copy and zero-allocation to provide high throughput and low latency.

Receiving process. The network transport pulls a buffer from the incoming buffer pool when new data can be received and fills it accordingly. The buffer is then pushed to the IBQ and processed by the **Message Creation Coordinator** thread (MCC) by deserializing the message headers. The message headers are pushed to the **message header store** afterwards. Multiple message handler threads concurrently create the message objects, deserialize the messages' payloads and pass the received Java objects to the application using its registered callback methods. When all data of a buffer has been processed, it is released and pushed back into the incoming buffer pool.

Incoming buffer pool. The buffer pool provides buffers, allocated in native memory, in different configurable sizes (e.g., 8×256 KB, 256×128 KB and 4096×16 KB). The transport pulls buffers using a worst-fit strategy as the amount of bytes ready to be received on the stream is unknown. It can also scale-up dynamically, if necessary.

The buffer pool management consists of three lock-free ring buffers optimized for access of one consumer and N producers (similar to the ORB but without the CUB, see Section IV).

A. Parallel Message Deserialization

Filled buffers are pushed by the transport thread into the IBQ. The IBQ is a basic ring buffer for one consumer and one producer and is synchronized using memory fences. The IBQ may be full and require the transport thread to park for a short moment and retry (see Section VII).

High throughput requires a parallel deserialization. As the received messages of the incoming stream can be split over several incoming buffers (see Section V-B), the buffer processing must be in-order and we need a two-staged approach to enable concurrency. The MCC thread pulls the buffer entries from the IBQ, deserializes all containing message headers (using relevant state information stored in the corresponding connection object) and pushes them into the message header store. Message payload deserialization based on the message headers can then be done in parallel by the message handler threads. This approach is efficient as the time-consuming payload deserialization and message object creation is parallelized.

The deserialization of split messages' payload (last message in buffer, which is not complete) must be in-order as well because all preceding parts of a message must be available to continue the deserialization of a split message. We address this situation by the MCC detecting and deserializing not only the header but the payload fraction within the current buffer, as well, for the split message. The rest of the message in the next buffer can be read by a message handler, again.

Split message headers are not a problem as deserialization of message headers is always done by the MCC which can store incomplete message headers within the connection object and continue with the next buffer.

Message header store. As mentioned before, the MCC pushes complete message headers to the message header store. The latter is implemented as a lock-free ring buffer for N consumers and one producer. Synchronization overhead is reduced by the MCC buffering the small message headers and pushing them in batches into the message header store. The batch size is limited but configurable, e.g., 25 headers.

Message header pool. Message headers are pooled, as well, in another single consumer, multiple producers lock-free ring buffer. Furthermore, message headers are pushed and pulled in batches. To reduce the probability of multiple message handler threads returning message headers at the same time, the batch sizes differ for every message handler.

Returning of buffers. A pooled buffer must not be returned before all its messages haven been deserialized. Because of the concurrent deserialization and split messages, we use the MCC incrementing an atomic counter for every message header pushed to the message header store (more precisely, the counter is increased once for every batch of message headers). Accordingly, the message handlers decrement the counter for every deserialized message. When all messages have been deserialized, the buffer can be safely returned to the pool.

We could run out of buffers during high throughput, if the MCC deserializes headers faster than the message handler threads can handle. Although we can scale up the number of incoming buffers, we prefer to throttle the MCC when a predefined number of used buffers is exceeded to reduce the memory consumption. Another benefit of limiting the amount of incoming buffers is that all buffer states like the message counters, the buffers' addresses or the unfinished operations which are filled for incomplete messages can be allocated once and reused for every incoming buffer to be processed.

Message Ordering. DXNet allows applications to mark messages and thus ensure message ordering on a stream/connection. All marked messages are guaranteed to be processed by the same message handler. All other steps preserve message ordering by default. For achieving maximum throughput, marking all messages is not advisable.

VII. Thread Parking Strategies

Lock-free programming allows low-latency synchronization but can easily overload a CPU by uncontrolled polling using CAS operations. DXNet implements a multi-level flow control with explicit message flow regulation and implicit throttling if memory pools drain and queues fill-up. We address three thread situations: blocked (the thread waits for another thread/server finishing its work because a pool is empty or queue full), colliding (failing CAS operation because another thread entered a critical section faster) and idling (the thread has nothing to do and waits for another thread/server committing new work).

Blocked thread. When blocked, the thread can park to reduce the CPU load because it is too fast compared to other threads/servers. However, the thread should not park for a long period to avoid restraining other threads/servers. Experiments showed that a sane park period is between 10 and 100 µs. Java allows minimum parking times of around 10 to 30 µs for a thread with `LockSupport.parkNanos()` for Linux servers with x86 CPUs.

Colliding thread. When colliding, the thread will repeat the CAS operation with updated values until successful because the thread is about to commit something and this should be done as fast as possible. However, reducing the collision probability (e.g., the ORB optimization described in IV-B) reliefs the CPU significantly.

Idling thread. This situation occurs, if a thread has nothing to do at the moment, e.g., a transport thread polls an empty ORB, the MCC polls an empty IBQ or a message handler polls an empty message header store. However, new work events can arrive within nanoseconds. Latency is minimized when threads do not park or yield, but only as long as the CPU is not overloaded. In case of CPU overload situations, parking threads can reduce latency.

We address this with an overprovisioning detection combined with an adaptive parking approach (10 to 30 µs), if the number of active threads (application threads and network threads) reaches a threshold, e.g., four times the number of cores, see also Section IX-A for the evaluation.

Idling for longer periods, e.g., applications not exchanging messages for a longer period of time, must be addressed, too. DXNet detects this, e.g., a network thread idling for one second (configurable time), and starts parking threads, if idling, reducing CPU load to a minimum.

VIII. Transport Implementations

DXNet has an open architecture supporting different network transport technologies. Currently, we have transport implementations for TCP/IP over Ethernet (using Java.nio), reliable verbs over Infiniband (based on JNI), and Loopback (for evaluation). Because of space constraints, we will only sketch some important aspects of these transports.

The `Ethernet` transport (EthDXNet) implementation is based on Java.nio and maps DirectByteBuffers to the ORB allowing to send data without copying it in user-space. Furthermore, two channels are opened for every connection to avoid channel duplication and for providing a side-band flow control channel for each connection. Channel duplication may occur when two servers create connections to each other simultaneously and must be avoided. The second channel allows exchanging flow control messages necessary to maximize throughput on a connection by using the back-channel.

The `InfiniBand` transport accesses the IBDXNet library (C++) using JNI. IBDXNet utilizes ibverbs to implement direct communication using the InfiniBand HCA. IBDXNet uses one dedicated send and one dedicated receive thread, both processing outgoing/incoming data in native memory. Context switching from C++ to Java was designed carefully and is highly optimized to avoid latency.

The `Loopback` transport is used for the experiments in this paper allowing to study the performance of DXNet without any bottlenecks from a real network. Data is not sent over a network device nor the operating system's loopback device (latency would be considerably high) but is directly copied from the ORB to a pooled incoming buffer. Furthermore, the Loopback transport simulates a server sending and receiving messages at highest possible throughput allowing to evaluate DXNet's performance.

IX. Evaluation

We evaluate the proposed concepts using Loopback and three different networks: 1 GBit/s Ethernet, 5 GBit/s Ethernet and 56 GBit/s InfiniBand. The Loopback is used to evaluate DXNet's concepts without any limitations of an underlying network.

Loopback and 5 GBit/s Ethernet tests were run in Microsoft's Azure cloud in Germany Central with up to 18 virtual machines from the type Standard_DS13_v2 which are memory optimized servers with 8 cores (Intel Xeon E5-2673), 56 GB RAM and shared 10 GBit/s Ethernet connectivity (two instances per connect). We deployed a custom Ubuntu 14.04 image with 4.4.0-59 kernel and Java 8. The tests with 1 GBit/s Ethernet and InfiniBand were executed on our private cluster servers with 64 GB RAM, Intel Xeon E5-1650 CPU and Ubuntu 16.04 with kernel 4.4.0-64.

We use a set of micro benchmarks for the evaluations in Sections IX-A and IX-B which send messages or requests of variable size with a configurable number of application threads. All throughput measurements refer to the payload size which is considerably smaller than the full message size, e.g., a 64-byte payload results in 115 bytes to be sent on IP layer when using Ethernet. Additionally, all runs with DXNet's benchmarks are **full-duplex** showing the aggregated performance for concurrently sending and receiving messages/requests.

A. Loopback

As mentioned before, we want to evaluate the efficiency of DXNet's concepts without any network limitations. Fig. 6 shows message processing times and throughputs for different message sizes when using the Loopback transport on a typical cloud server (Standard_DS13_v2). Messages up to 2 KB can be processed in around 500 ns. Larger messages require

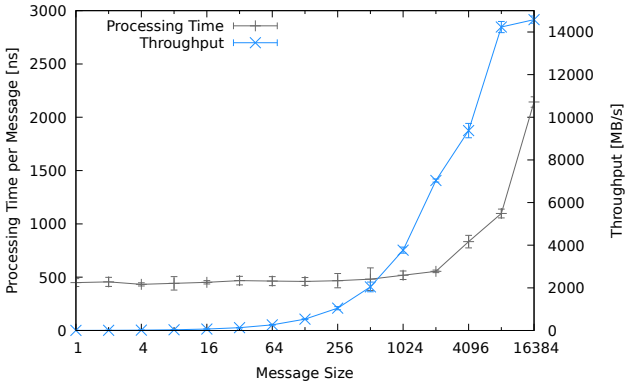

Figure 6. 10^7 Messages, 1 App. Thread, 4 Message Handlers.

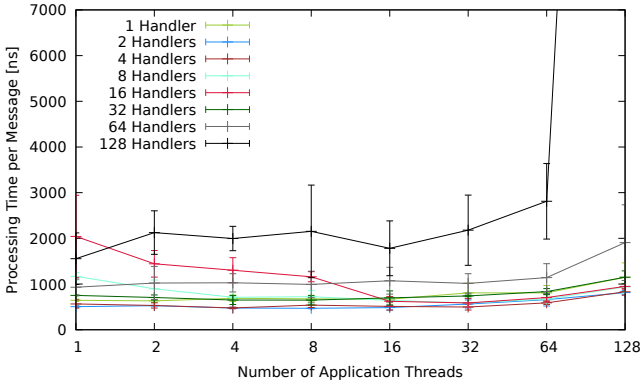

Figure 7. 10^7 Messages, 1024 Bytes Payload.

increasing processing times, as expected. The throughput increases linearly with the message size up to 8 KB messages and is capped at around 14 GByte/s aggregated throughput for sending and receiving of larger messages. The Linux tool `mbw` determined a memory bandwidth of 7.19 GByte/s for a 16 GB array and 16 KB block for the used servers which explains the maximum throughput (saturation of the available memory bandwidth).

In Fig. 6, we studied messages with up to 16 KB payload size as DXNet is primarily designed to perform well with small messages. We also tested larger messages (larger than the ORB, configured with 4 MB here) and measured a message throughput of around 5.4 GByte/s with 8 MB messages. The throughput is lower as application threads and transport thread work sequentially for larger messages (see Section V-A). However, if the application needs to often handle large messages, throughput can easily be improved by using a larger ORB.

DXNet is designed to efficiently support concurrent application threads sending and receiving messages in parallel. Fig. 7 shows that the processing time for 1 KB messages is stable from one to 64 and only slightly increases with 128 application threads. Additionally, Fig. 7 shows the performance with a varying number of message handlers peaking with two to four. Obviously, 128 application threads and 128 message handlers overstress the CPU (8 cores) significantly. The results for all other constellations are as expected showing DXNet's capability of efficiently handling hundreds of concurrent threads.

We also evaluated request-response latency by measuring the **RTT**, which includes sending a request, receiving the

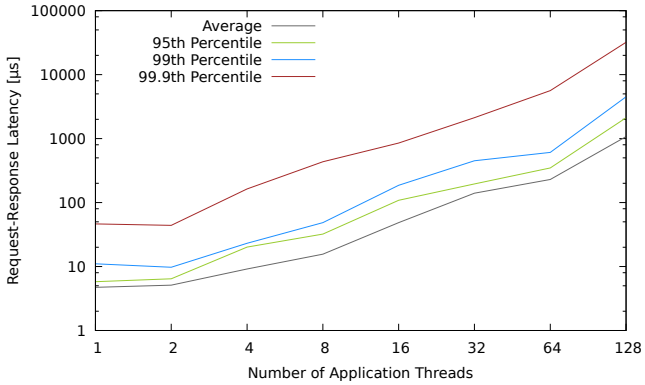

Figure 8. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

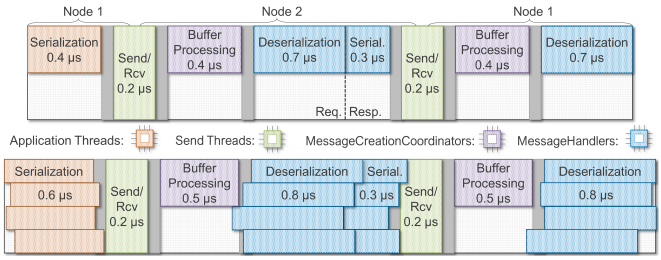

Figure 9. Breakdown of Request-Response Latency for 1024-byte Requests. One application thread (on top) and four (at the bottom). Grey bars indicate inter-thread communication.

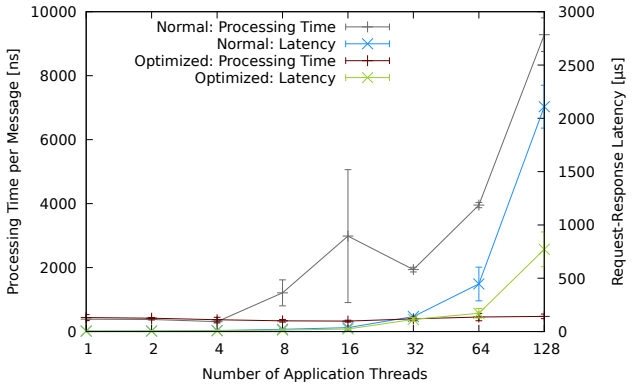

Figure 10. 10^7 Message or 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

request, sending the corresponding response and receiving the response. Fig. 8 shows the latency for small requests with increasing number of application threads. The average RTT with one and two application threads is under 5 µs. With up to eight threads the RTT increases slower than the number of threads because requests can be aggregated for sending. With more threads the increase rate is higher.

Fig. 9 shows the breakdown of request-response latency for one and four application threads and 1024-byte requests. This is a best-effort approximation as time measurement is costly and influences the processing. As expected de-/serialization accounts for the majority of the RTT and deserialization is slower than serialization because of the message object allocation and creation. With more application threads or asynchronous messages, all depicted steps are executed in parallel.

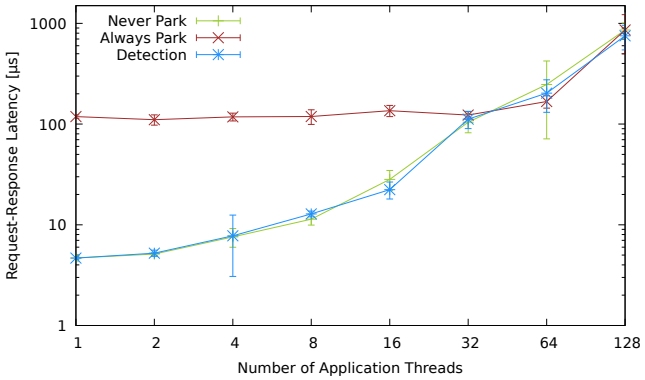

Figure 11. 10^6 Requests, 2 Message Handlers, 1 Byte Payload.

Optimized Outgoing Ring Buffer. The benefits of the CUB, discussed in Section IV, can be seen in Fig. 10. Without the optimization the message processing time increases significantly with more than four application threads sending messages (with 128 threads nearly 20 times higher). Furthermore, the RTT diverges considerably with more than 32 application threads as well.

Overprovisioning Detection. Fig. 11 shows the importance of the thread parking strategy (see Section VII). The RTT is 25 times larger when using one application thread and always parking network threads. All three strategies match with 32 threads and diverge a little with more threads. The never park strategy is at disadvantage with many threads (128) and the RTT is around 100 µs larger than with the adaptive approach.

The evaluation with Loopback transport shows the high throughput and low latency of DXNet. Furthermore, DXNet offers a high stability when used with many threads sending and receiving messages in parallel.

B. Comparing Network Transports

Fig. 12 shows the message processing time and throughput for all three network transports (Ethernet and Loopback on cluster and cloud instances) with varying payload size. As expected, InfiniBand has the lowest processing overhead and highest throughput of all physical devices.

The comparison between the 1 GBit/s Ethernet of the private cluster and 5 GBit/s Ethernet in Azure cloud reveals interesting insights. Obviously, message throughput is higher in the cloud for large messages. But, message throughput is higher and processing time is lower on the cluster for messages smaller than 64 bytes which is most likely caused by the virtualization overhead of cloud servers. Loopback is also considerably faster on cluster instances (< 300 ns processing time and > 16 GByte/s throughput).

Fig. 13 shows the request-response latency and throughput for requests sent by four application threads. Again, 1 GBit/s Ethernet on our cluster performs better for small payloads (< 1024) than 5 GBit/s Ethernet in the cloud. For larger requests the bandwidth becomes more and more important favoring the cloud network. Both Ethernet networks are far off the latencies InfiniBand achieves. For small request (< 512 byte payload) the RTT is consistently under 10 µs and rises to only 16 µs for 16 KB requests. Hence, the throughput is much higher with InfiniBand as well.

The evaluation with three physical transports confirms the

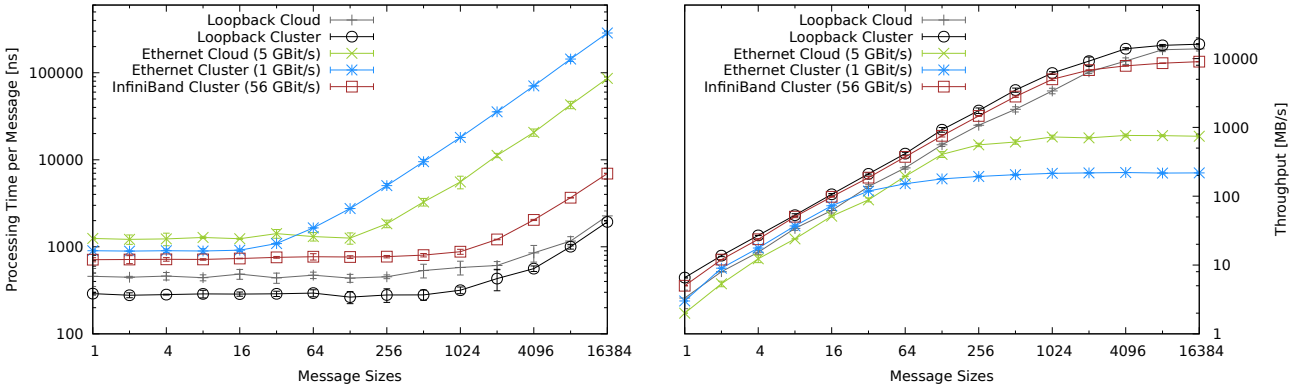

Figure 12. 10^8 Messages, 1 App. Thread, 2 Message Handlers.

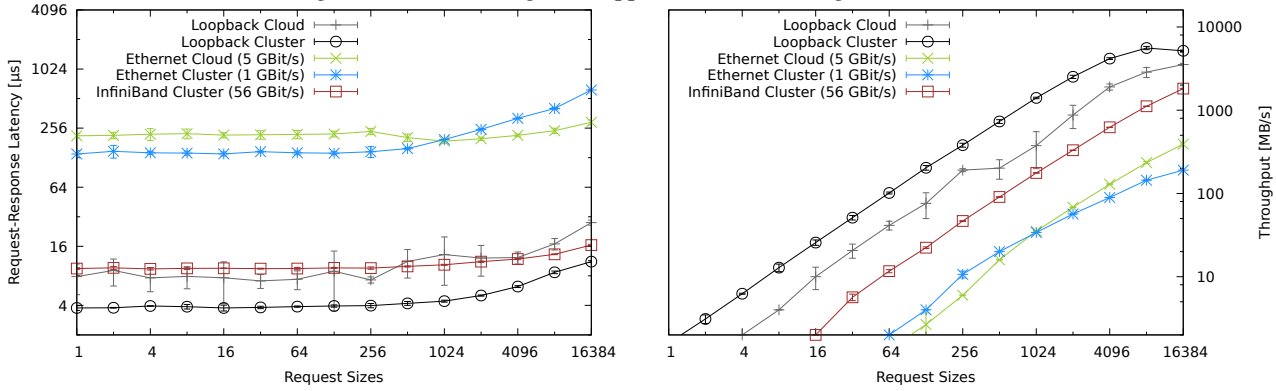

Figure 13. 10^7 Requests, 4 App. Threads, 2 Message Handlers.

results gathered with Loopback and DXNet performs strong especially with InfiniBand (RTT < 10 µs, throughput > 9 GByte/s full-duplex).

C. Yahoo! Cloud Serving Benchmark

The Yahoo! Cloud Serving Benchmark (YCSB) was designed to quantitatively compare distributed serving storage systems [33]. The benchmark offers a set of simple operations (reads, writes, range scans) and a tabular key-value data model to evaluate online storage systems regarding their elasticity, availability and replication. Furthermore, YCSB is easily extensible for new storage systems and new workloads. For our evaluation, we used the in-memory key-value store DXRAM [34] which utilizes DXNet and created an individual workload: one 64-byte object per key, 10^6 keys, uniform distribution, 90 % read and 10 % write operations, 10^7 operations. The tests were run in the Microsoft Azure cloud with one storage server and an increasing number of client servers (maximum 16) which each hosted up to 80 client threads.

Fig. 14 shows the average operation latency and throughput with 10 to 1280 client threads. The operation latency starts at around 230 µs which is in line with previous latency measurements. The latency grows slowly up to 480 client threads but then exponentially indicating server congestion. The throughput rises up to 640 client threads with more than one million operations per second and remains stable with more client threads.

The evaluation with YCSB shows DXNet's high performance for a client-server scenario (one server can serve more than 1000 clients).

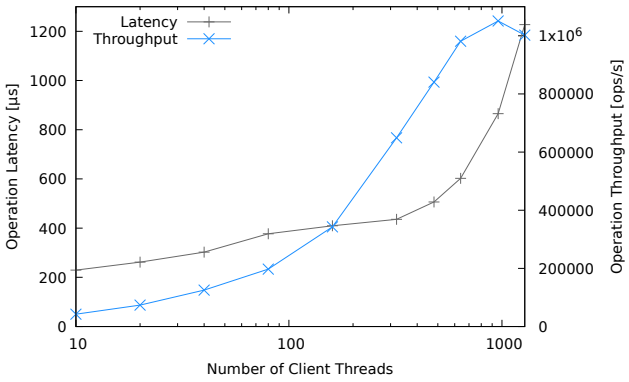

Figure 14. 6 Message Handlers

X. Conclusions

Many big data applications as well as large scale interactive applications are written in Java and aggregate the resources of many servers in a cloud data center, high performance cluster or private cluster. Efficient network communication is very important for these application domains. RMI while being comfortable to use is not fast enough for these applications. Plain sockets are difficult to handle especially if efficiency and scalability need to be addressed. MPI was designed for spawning processes with finite runtime in a static environment. Thus, multi-threading performance and support for adding/removing nodes to an existing environment are limited.

In this paper, we proposed DXNet, a Java open-source network library complementing the communication spectrum.

DXNet provides fast parallel serialization for Java objects, automatic connection management, automatic message aggregation and an event-driven message receiving approach including a concurrent deserialization. DXNet offers high-throughput asynchronous messaging as well as synchronous request/response communication with very low latency. Finally, its architecture is open for supporting different transport protocols. It already supports TCP with Java.nio and reliable verbs for Infiniband. DXNet achieves high performance and low latency by using lock-free data structures, zero-copy and zero-allocation. The proposed ring buffer and queue structures are complemented by different thread parking strategies guaranteeing low latency by avoiding CPU overload.

Evaluations on a private cluster and in the Microsoft Azure cloud show message processing times of sub 300 ns resulting in throughputs of up to 16 GByte/s which saturate the memory bandwidth of a typical cloud instance. For the request/response pattern, DXNet is able to provide sub 10 µs RTT latency using the InfiniBand transport (sub 4 µs over Loopback). Finally, DXNet is also able to efficiently handle highly concurrent processing of many small messages resulting in throughput saturations for Ethernet with 256 bytes payload and InfiniBand with 1-2 KB payload.

The InfiniBand transport IBDXNet is work in progress and final results will be published separately (throughput: >10.4 GByte/s). Future work also includes more experiments at larger scales including comparisons with other network middlewares, as well as evaluations using a 100 GBit/s InfiniBand network.

REFERENCES

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, pp. 1804–1815, Aug. 2015.

[2] M. S. Engler, M. El-Kebir, J. Mulder, A. E. Mark, D. P. Geerke, and G. W. Klau, "Enumerating common molecular substructures," *PeerJ Preprints*, vol. 5, p. e3250v1, Sep. 2017.

[3] P. Satapathy, J. Dave, P. Naik, and M. Vutukuru, "Performance comparison of state synchronization techniques in a distributed lte epc," in *IEEE Conf. on Network Function Virtualization and Software Defined Networks*, 2017.

[4] S. Ekanayake, S. Kamburugamuve, and G. C. Fox, "Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium*, 2016, pp. 3:1–3:8.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[6] "Cassandra," http://cassandra.apache.org, accessed: 2018-03-14.

[7] "Interactive query with apache hive on apache tez," http://hortonworks.com/hadooptutorial/supercharging-interactivequeries-hive-tez/, accessed: 2018-03-14.

[8] "Impala - cloudera," https://www.cloudera.com/products/open-source/apache-hadoop/impala.html, accessed: 2018-03-14.

[9] S. Microsystems, "Java remote method invocation specification," https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, accessed: 2018-03-14.

[10] Oracle, "Package java.net," https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html, accessed: 2018-03-14.

[11] S. Mintchev, "Writing programs in javampi," University of Westminster, Tech. Rep. MAN-CSPE-02, Oct. 1997.

[12] K. Beineke, S. Nothaas, and M. Schoettner, "Dxnet project on github," https://github.com/hhu-bsinfo/dxnet, accessed: 2018-03-14.

[13] W. Zhu, C.-L. Wang, and F. C. M. Lau, "Jessica2: a distributed java virtual machine with transparent thread migration support," in *Proceedings. IEEE International Conference on Cluster Computing*, 2002, pp. 381–388.

[14] S. P. Ahuja and R. Quintao, "Performance evaluation of java rmi: A distributed object architecture for internet based applications," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '00, 2000, pp. 565–569.

[15] M. Philippsen, B. Haumacher, and C. Nester, "More efficient serialization and rmi for java," *Concurrency: Practice and Experience*, vol. 12, pp. 495–518, 2000.

[16] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman, "Efficient java rmi for parallel programming," *ACM Trans. Program. Lang. Syst.*, vol. 23, pp. 747–775, Nov. 2001.

[17] C. Nester, M. Philippsen, and B. Haumacher, "A more efficient rmi for java," in *Proc. of the ACM 1999 Conf. on Java Grande*, 1999, pp. 152–159.

[18] M. P. I. Forum, Ed., *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center, 2015, 2015. [Online]. Available: https://books.google.de/books?id=Fbv7jwEACAAJ

[19] A. Shafi, B. Carpenter, and M. Baker, "Nested parallelism for multi-core hpc systems using java," in *Journal of Parallel and Distributed Computing*, 2009, pp. 532–545.

[20] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and X. Li, "mpijava: A java interface to mpi," http://www.hpjava.org/mpiJava.html, accessed: 2018-03-14.

[21] G. "Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded mpi communication on multicore petascale systems," in *"Recent Advances in the Message Passing Interface"*, 2010, pp. 11–20.

[22] H. V. Dang, S. Seo, A. Amer, and P. Balaji, "Advanced thread synchronization for multithreaded mpi implementations," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 314–324.

[23] R. Latham, R. Ross, and R. Thakur, "Can mpi be used for persistent parallel services?" in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2006, pp. 275–284.

[24] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using mpi in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, 2013, pp. 43–48.

[25] K. Beineke, S. Nothaas, and M. Schoettner, "Fast parallel recovery of many small in-memory objects," in *International Conference on Parallel and Distributed Systems (ICPADS)*, vol. 23, in press.

[26] Oracle, "Java i/o, nio, and nio.2," https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html, accessed: 2018-03-14.

[27] R. Hitchens, *Java NIO*. Sebastopol, CA, USA: O'Reilly Media, 2009.

[28] G. L. Taboada, J. Touriño, and R. Doallo, "Java fast sockets: Enabling high-speed java communications on high performance clusters," *Comput. Commun.*, vol. 31, pp. 4049–4059, Nov. 2008.

[29] G. L. Taboada, J. Tourino, and R. Doallo, "High performance java sockets for parallel computing on clusters," in *Parallel and Distributed Processing Symposium*, 2007, pp. 1–8.

[30] L. Mastrangelo, L. Ponzanelli, A. Mocci, M. Lanza, M. Hauswirth, and N. Nystrom, "Use at your own risk: The java unsafe api in the wild," *SIGPLAN Not.*, vol. 50, pp. 695–710, Oct. 2015.

[31] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat, "Pickling state in the javatm system," in *Proc. of the 2nd Conf. on USENIX Conf. on Object-Oriented Technologies*, 1996, pp. 19–19.

[32] "Kryo - java serialization and cloning: fast, efficient, automatic." https://github.com/EsotericSoftware/kryo, accessed: 2018-03-14.

[33] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[34] K. Beineke, S. Nothaas, and M. Schoettner, "High throughput log-based replication for many small in-memory objects," in *IEEE 22nd International Conference on Parallel and Distributed Systems*, 2016, pp. 535–544.

Chapter 3.

Scalable Messaging for Java-based Cloud Applications

This chapter summarizes the contributions and includes a copy of our paper [18].

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Scalable Messaging for Java-based Cloud Applications". In: ICNS 2018, The Fourteenth International Conference on Networking and Services. May 2018, pp. 32-41

Selected to be published as an extended version in the International Journal On Advances in Internet Technology, v 11, 2018. To be submitted.

3.1. Paper Summary

In Chapter 2 (based on [13]), we describe the messaging network system DXNet which is transport agnostic and supports Ethernet and InfiniBand interconnect. In this publication, we detail EthDXNet, the Ethernet transport implementation of DXNet. Our approach is based on Java.nio's socket channel and selector and provides a low-overhead interest handling and scalable automatic connection management. In EhtDXNet, every connection consists of two socket channels, one for incoming data and one for outgoing data, to use the back-channels of both socket channels for out-of-band data like the application-level flow control of DXNet. Additionally, using separate socket channels for incoming and outgoing traffic avoids channel duplication, a well-known problem occurring when two servers create a channel to each other simultaneously. Furthermore, EthDXNet uses direct ByteBuffers which can be copied into kernel socket buffers directly, avoiding copying the buffers in user-space.

The evaluation shows the low latency and high throughput as well as the good scalability of DXNet and EthDXNet even in an all-to-all communication pattern (worst case) with up to 64 servers in the Microsoft Azure cloud (each server connected with 5 Gbit/s Ethernet which is a shared 10 GBit/s Ethernet connection).

3.2. Importance and Impact on Thesis

Ethernet is one of the most commonly used interconnects in data centers and clusters, even in HPCs: 248 of the top 500 supercomputers are connected with Ethernet (204 using 10 GBit/Ethernet) [112]. Microsoft uses 10 GBit/s Ethernet for connecting the servers in the Azure cloud data centers in Germany as well. Therefore, providing an Ethernet transport implementation for DXNet and DXRAM is imperative to exploit new applications. For this thesis, an earlier version of the Ethernet implementation, which shared many aspects described in this publication, was used for the evaluation in [15] and [14].

3.3. Personal Contribution

This paper describes the Ethernet transport of DXNet (EthDXNet). Therefore, the contributions of the DXNet paper are also applicable to this paper. However, all detailed contributions in sections IV to VII were developed by Kevin Beineke, the author of this thesis. Stefan Nothaas aided by revising and upgrading the DXNet benchmark for multi-node communications. Furthermore, Stefan Nothaas optimized DXNet's flow control to improve his InfiniBand transport which was also applied to EthDXNet by Kevin Beineke.

Kevin Beineke structured the publication and wrote most of it. Stefan Nothaas contributed by writing an outline of the DXNet paper (Section III) and proof-reading the paper. Prof. Dr. Michael Schöttner participated in discussions regarding the proposed approaches and evaluation and also reviewed the paper.

Scalable Messaging for Java-based Cloud Applications

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—Many big data and large-scale cloud applications are written in Java or are built using Java-based frameworks. Typically, application instances are running in a data center on many virtual machines which requires scalable and efficient network communication. In this paper, we present the practical experience of designing a Java.nio transport for DXNet, a low latency and high throughput messaging system which goes beyond message passing by providing a fast parallel object serialization. The proposed design uses a zero-copy send and receive approach avoiding copying data between de-/serialization and send/receive. It is based on Java.nio socket channels complemented by application-level flow control to achieve low latency and high throughput for >10 GBit/s Ethernet networks. Furthermore, a scalable automatic connection management and a low-overhead interest handling provides an efficient network communication for dozens of servers, even for small messages (< 100 bytes) and all-to-all communication pattern. The evaluation with micro benchmarks shows the efficiency and scalability with up to 64 virtual machines in the Microsoft Azure cloud. DXNet and the Java.nio-based transport are open source and available on Github.**

Keywords–Message passing; Ethernet networks; Java; Data centers; Cloud computing;

I. Introduction

Big data processing is emerging in many application domains whereof many are developed in Java or are based on Java frameworks [1][2][3]. Typically, these big data applications aggregate the resources of many virtual machines in cloud data centers (on demand). For data exchange and coordination of application instances, an efficient network transport is very important. Fortunately, public cloud data centers already provide 10 GBit/s Ethernet and faster.

Java applications have different options for exchanging data between Java servers, ranging from high level Remote Method Invocation (RMI) [4] to low-level byte streams using Java sockets [5] or the Message Passing Interface (MPI) [6]. However, none of the mentioned possibilities offer high performance messaging, elastic automatic connection management, advanced multi-threaded message handling and object serialization all together. Therefore, we proposed DXNet [7], a network messaging system which meets all of these requirements. DXNet is extensible by transport implementations to support different network interconnects.

In this paper, we propose an Ethernet transport implementation for DXNet, called EthDXNet. The transport is based on Java.nio and provides high throughput and low latency networking over Ethernet connections.

The contributions of this paper are:

- the design of EthDXNet and practical experiences including:
  - scalable automatic connection management
  - zero-copy approach for sending and receiving data over socket channels
  - efficient interest handling
- evaluations with up to 64 VMs in the Microsoft Azure cloud

The evaluation shows that EthDXNet scales well while per-node message throughput and request-response latency is constant from 2 to 64 nodes, even in an high-load all-to-all scenario (worst case). Furthermore, high throughput is guaranteed for small 64-byte messages and saturation of the physical network bandwidth (5 GBit/s) with 4 KB messages. The latency experiments also show that EthDXNet efficiently utilizes the underlying network as long as the CPU does not get overstressed by too many application threads leading to a natural increase in latency.

The structure of the paper is as follows: after discussing related work, we present an overview of DXNet in Section III. In Section IV, we describe the sending and receiving procedure of EthDXNet, followed by a presentation of the connection management in Section V. Section VI focuses on the flow control implementation and Section VII on the interest handling. The evaluation is in Section VIII. The conclusions can be found in Section IX.

II. Related Work

In this section, we discuss the related work for this paper.

A. JavaRMI

Java's RMI [4] provides a high level mechanism to transparently invoke methods of objects on a remote machine, similar to Remote Procedure Calls (RPC). Parameters are automatically de-/serialized and references result in a serialization of the object itself and all reachable objects (transitive closure), which can be costly. Missing classes can be loaded from remote servers during RMI calls, which is very flexible but introduces even more complexity and overhead. Additionally, the built-in serialization is known to be slow and not very

space efficient [8][9]. Furthermore, method calls are always blocking.

B. MPI

MPI is the state-of-the-art message passing standard for parallel high performance computing. It is available for Java applications by implementing the MPI standard in Java or wrapping a native library. However, MPI was designed for spawning jobs with finite runtime in a static environment. DXNet's and EthDXNet's main application domain are ongoing applications with dynamic node addition and removal (not limited to). The MPI standard defines the required functionality for adding and removing processes, but common MPI implementations are incomplete in this regard [10][11]. Furthermore, job shutdown and crash handling is still improvable [11].

C. Java.nio

The `java.io` and `java.net` libraries provide basic implementations for exchanging data via TCP/IP and UDP sockets over Input- and OutputStreams [12][5]. To create a TCP/IP connection between two servers, a new `Socket` is created and connection established to a remote IP and port. On the other end, a `ServerSocket` must be listening on given IP-port tuple creating a new Socket when accepting an incoming connection-creation request. The connection creation must be acknowledged from both sides and can be used to exchange byte arrays by reading/writing from/to the Socket hereafter. While this is sufficient for small applications with a few connections, this basic approach lacks several performance-critical optimizations [13] introduced with `Java.nio` [12][14]. (1) Instead of byte arrays, the read/write methods of Java.nio use `ByteBuffers`, which provide efficient conversion methods for all primitive data types. (2) ByteBuffers can be allocated outside of the Java heap allowing system-level I/O operations on the data without copying as the ByteBuffer is not subject to the garbage collection outside of the Java heap. Obviously, this relieves the garbage collector as well lowering the overhead with many buffers. (3) `SocketChannels` and `Selectors` enable asynchronous, non-blocking operations on stream-based sockets. With simple Java Sockets, user-level threads have to poll (a blocking operation) in order to read data from a Socket. Furthermore, when writing to a Socket the thread blocks until the write operation is finished, even if the Socket is not ready. With Java.nio, operation interests (like READ or WRITE) are registered on a Selector which selects operations when they are ready to be executed. This enables efficient handling of many connections with a single thread. The dedicated thread is required to call the `select` method of the Selector which is blocking if no socket channel is ready or returns with the number of executable operations. All available operations (e.g., sending/receiving data) can be executed by the dedicated thread, afterwards.

D. Java Fast Sockets

Java Fast Sockets (JFS) is an efficient Java communication middleware for high performance clusters [15]. It provides the widely used socket API for a broad range of target applications and is compatible with standard Java compilers and VMs. JFS avoids primitive data type array serialization (JFS does not include a serializer), reduces buffering and unnecessary copies in the protocol and provides shared memory communication with an optimized transport protocol for Ethernet. DXNet provides a highly concurrent serialization for complex Java objects and primitive data types which avoids copying/buffering as well. EthDXNet is an Ethernet transport implementation for DXNet.

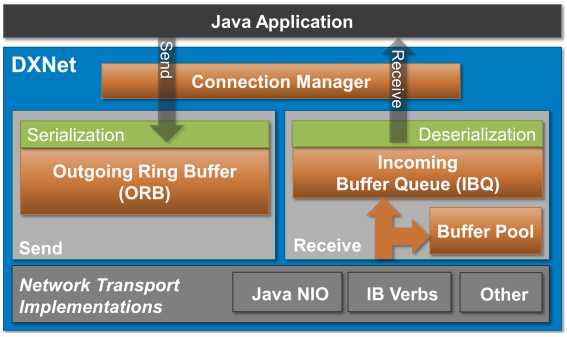

Figure 1. Simplified DXNet Architecture (from [7])

III. DXNet

DXNet is a network library for Java targeting, but not limited to, big data applications. DXNet implements an **event driven message passing** approach and provides a simple and easy to use application interface. It is optimized for highly multi-threaded sending and receiving of small messages by using **lock-free data structures, fast concurrent serialization, zero copy and zero allocation**. Split into two major parts, the core of DXNet provides automatic connection management, serialization of message objects and an interface for implementing different transports. Currently, an Ethernet transport using Java.nio sockets and an InifiniBand transport using ibverbs is implemented.

This section describes the most important aspects of DXNet and its core (see Figure 1) which are relevant for this paper. However, a more detailed insight of the core is given in a dedicated paper [7]. The source code is available at Github [16].

A. Connection Management

All nodes are addressed using an **abstract 16-bit node ID**. Address mappings must be registered to allow associating the node IDs of each remote node with a corresponding implementation dependent endpoint (e.g., socket, queue pair). To provide scalability with up to hundreds of simultaneous connections, our event driven system does not create one thread per connection. A **new connection is created automatically** once the first message is either sent to a destination or received from one. Connections are closed once a configurable connection limit is reached using a recently used strategy. Faulty connections (e.g., remote node not reachable anymore) are handled and cleaned up by the manager. Error handling on connection errors or timeouts are propagated to the application using exceptions.

B. Sending of Messages

Messages are Java objects and sent asynchronously. A message can be targeted towards one or multiple receivers. Using the message type **Request**, it is sent to one receiver. The

sender waits until receiving a corresponding response message (transparently handled by DXNet) or skips waiting and collects the response later.

One or multiple application threads call DXNet (concurrently) to send messages. Every message is automatically and concurrently serialized into the **Outgoing Ring Buffer (ORB)**, a natively allocated and lock-free ring buffer. When used concurrently, messages are automatically aggregated which increases send throughput. The ORB, one per connection, is allocated in native memory to allow direct and zero-copy access by the low level transport. The transport runs a decoupled dedicated thread which removes the serialized and ready to send data from the ORB and forwards it to the hardware.

C. Receiving of Messages

The network transport handles incoming data by writing it to **pooled native buffers**. We use native buffers and pooling to avoid burdening the Java garbage collection. Depending on how a transport writes and reads data, the buffer might contain fully serialized messages or just fragments. Every buffer is pushed to the ring buffer based **Incoming Buffer Queue (IBQ)**. Both, the buffer pool as well as the IBQ are shared among all connections. Dedicated **message handler threads** pull buffers from the IBQ and process them asynchronously by de-serializing them and creating Java message objects. The messages are passed to pre-registered callback methods of the application.

D. Flow Control

DXNet implements its own **flow control (FC)** mechanism to avoid flooding a remote node with messages. This would result in an increased overall latency and lower throughput if the remote node cannot keep up with processing incoming messages. When sending messages, the per connection dedicated FC checks if a configurable threshold is exceeded. This threshold describes the **number of bytes sent by the current node but not fully processed by the receiving node**. The receiving node counts the number of bytes received and sends a confirmation back to the source node in regular intervals. Once the sender receives this confirmation, the number of bytes sent but not processed is reduced. If an application send thread was previously blocked due to exceeding this threshold, it can now continue with processing the message.

E. Transport Interface

DXNet provides a transport interface allowing implementations of different transport types. One of the implemented transports can be selected on the start of DXNet. The transport and its specific semantics are transparent to the applications.

The following methods must be implemented for every transport:

- Setup connection
- Close and cleanup connection
- Signal to send data available in the ORB of a connection (callback)
- Pull data from the ORB and send it
- Push received raw data/buffer to the IBQ

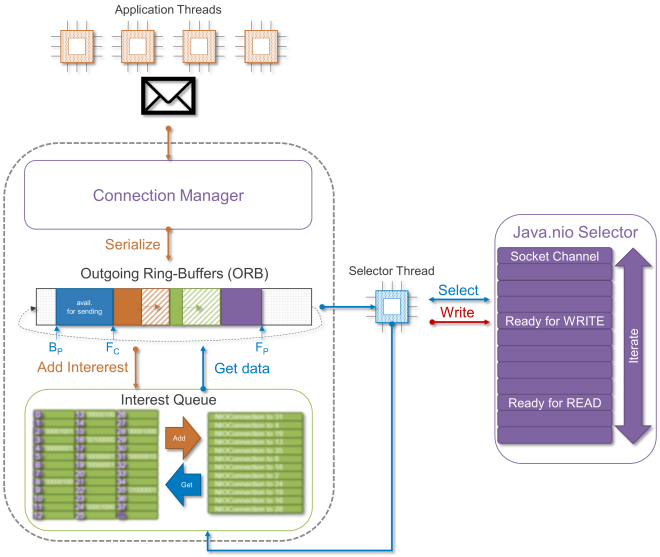

Figure 2. Data structures and Threads. Details of the Interest Queue can be found in Figure 4.

IV. EthDXNet - Sending and Receiving

In the following sections, we describe the Ethernet transport of DXNet, called EthDXNet. An overview of the most important data structures and threads of EthDXNet are depict in Figure 2.

A. Sending of Data

To send messages, the DXNet API methods `sendMessage` or `sendSync` are called by the application threads (or message handler threads). In DXNet, messages are always sent asynchronously, i.e., application threads might return before the message is transfered. It is possible, though, to wait for a response before returning to the application (sendSync). After getting the `ConnectionObject` (a Java object) from the **Connection Manager**, the message is serialized into the ORB associated with the connection. For performance reasons, many application threads can serialize into the same or different ORBs in parallel (more in [7]). The actual message transfer is executed by the **SelectorThread**, a dedicated daemon thread driving the Java.nio back-end. Thus, after serializing the message into the ORB, the application thread must signal data availability for the corresponding connection. This is done by registering a `WRITE` interest (see Table I) for given connection in the **Interest Queue** (see Section VII). When ready, Java.nio's **Selector** wakes-up the SelectorThread (which is blocked in the `select` method of the Selector) to execute the operation and thus transferring the message.

After returning from the select method, a **SelectionKey** is available in the ready-set of the Selector. It contains the operation interest WRITE, the socket channel and attachment (the associated ConnectionObject). This SelectionKey is dispatched based on the operation. In order to send the message over the network, the SelectorThread pulls the **data block** from the ORB of the corresponding connection and calls the `write` method of the socket channel. From this point, we cannot distinguish single messages anymore because

messages are naturally aggregated to data blocks in the ORBs, which is a performance critical aspect. The write method is called repeatedly until all bytes have been transferred or the method returned with return value 0. The second case indicates congestion on the network or the receiver and is best handled by stopping the transfer and continue it later. After sending, the back position (B_p, see Figure 2) of the ORB is moved by the number of bytes transferred to free space for new messages to send. Additionally, if the transfer was successful and the ORB is empty afterwards, the SelectionKey's operation is set to `READ` which is the preset operation and enables receiving incoming data blocks. If the transfer failed, the connection is closed (see Section V). If the transfer was incomplete or new data is available in the ORB, the SelectionKey is set to `READ | WRITE` (combination of READ and WRITE by using the bitwise or-operator) which triggers a new WRITE operation when calling select the next time but also allows receiving incoming messages. It is important to change the SelectionKey to this state as keeping only the WRITE operation could result in a deadlock situation in which both ends try to transfer data but none of them are able to receive data on the same connection. This causes the **kernel socket receive buffers** to fill up on both sides preventing further data transfer.

The ORB is a ring buffer allocated in native memory (outside of the Java heap). In order to pass a ByteBuffer to the socket channel, which is required for data transfer, we wrap a **DirectByteBuffer** onto the ORB and set the ByteBuffer's position to the front position in the ORB and the limit to the back position. A DirectByteBuffer is a ByteBuffer whose underlying byte array is stored in native memory and is not subject to garbage collection. This enables native operations of the operating system without copying the data first. The socket channel's send and receive operations are examples for those native operations, thus, benefiting from the DirectByteBuffer. Java does not support changing the address of a ByteBuffer. Therefore, on initialization of the ORB, we allocate a new DirectByteBuffer by calling `allocateDirect` of the Java object ByteBuffer and use the underlying byte array as the ORB. To do so, we need to determine the memory address of the byte array, which can be obtained with `Buffer.class.getDeclaredField("address")`. That is, during serialization the ORB is accessed with `Java.unsafe` by reading/writing from/to the actual address outside of the Java heap, but the socket channel accesses the data by using the DirectByteBuffer's reference (with adjusted position and limit). We do not access the ORB by using the DirectByteBuffer during serialization because of performance and compatibility reasons described in [7].

Although this approach prevents copying the data to be sent on user-level, the data is still copied from the ORB to the **kernel socket send buffer** which is a necessity of the stream-based socket approach. Therefore, configuring the kernel socket buffer sizes (one for sending and one for receiving) correctly has a great impact on performance. We empirically determined setting both buffer sizes to the ORBs' size offers a good performance without increasing the memory consumption too much (typically the ORBs are between 1 and 4 MB depending on the application use case).

B. Receiving of Data

Receiving messages is always initiated by Java.nio's **Selector** which detects incoming data availability on socket channels. When a socket channel is ready to be read from, the SelectorThread selects the SelectionKey and dispatches the READ operation. Next, the SelectorThread reads repeatedly by calling the `read` operation on the socket channel until there is nothing more to read or the buffer is full. If reading from the socket channel failed, the socket channel is closed. Otherwise, the ByteBuffer with the received data is flipped (limit = position, position = 0) and pushed to the IBQ (see Figure 1). The buffer processing is explained in [7].

In order to read from the socket channel, a ByteBuffer is required to write the incoming data into. Constantly allocating new ByteBuffers decreases the performance drastically. Therefore, we implemented a **buffer pool**. The buffer pool provides ByteBuffers, allocated in native memory (which are DirectByteBuffers), in different configurable sizes (e.g., 8×256 KB, 256×128 KB and 4096×16 KB). The SelectorThread pulls DirectByteBuffers using a worst-fit strategy as the amount of bytes ready to be received on the stream is unknown. It can also scale-up dynamically, if necessary. The buffer pool management consists of three lock-free ring buffers optimized for access of one consumer and N producers [7].

The pooled DirectByteBuffers are wrapped to provide the ByteBuffer's reference as well as the ByteBuffer's address. The reference is used for reading from the socket channel and the address is necessary to deserialize the messages within the ByteBuffer.

V. Automatic Connection Management

For sending and receiving messages, we have to manage all open connections and create/close connections on demand. A connection is represented by an object (ConnectionObject), containing a node ID to identify the connection based on the destination, a **PipeIn** and a **PipeOut**. The PipeOut consists mostly of an ORB, a socket channel and flow control for outgoing data. The PipeIn contains a socket channel, flow control for incoming data, has access to the buffer pool (shared among all connections) and more data structures important to buffer processing, which are not further discussed in this paper.

1) Connection Establishment: Connections are created in two ways: (1) actively by creating a new connection to a remote node or (2) passively by accepting a remote node's connection request. In both cases, the connection manager must be updated to administrate the new connection. Figure 3 shows the procedure of creating a new connection (active on the left side and passive on the right). The core part is the TCP handshake, which can be seen in the middle.

Active connection creation: A connection is created actively, if an application thread wants to send a message to a not yet connected node. To establish the connection, the application thread creates a new ConnectionObject (including PipeIn and PipeOut and all its components), opens a new socket channel and connects the socket channel to the remote node's IP and port. Afterwards, the application thread registers a `CONNECT` operation, creates a `ReentrantLock` and `Condition` and waits until the Condition is signaled or

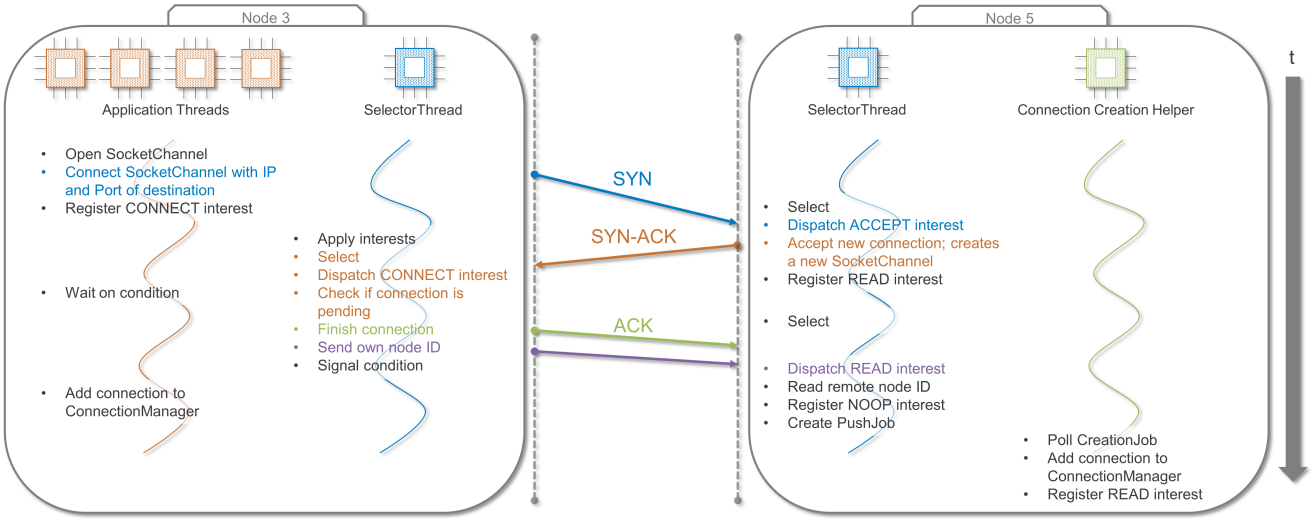

Figure 3. Connection Creation

the connection creation was aborted. To correctly identify the corresponding ConnectionObject to a socket channel, the ConnectionObject is attached to the SelectionKey when registering the CONNECT interest and all following interests.

The SelectorThread continues the connection establishment by applying the CONNECT interest and selecting the socket channel when the remote node accepted the connection or the connection establishment failed. After selecting the SelectionKey, the socket channel's status is checked. If it is pending, the connection creation was successful so far and the socket channel can be completed by calling `finishConnect`. If the connection establishment was aborted, the application thread is informed by setting a flag (which is checked periodically by the application thread).

The remote node has to identify the new node currently creating a connection. Thus, the node ID is sent to the remote node on the newly created channel. Furthermore, the SelectorThread marks the PipeOut as connected and signals the condition so the application thread can continue. The application thread adds the connection to the connection manager, increments the connection counter and starts sending data, afterwards.

Passive connection creation: For accepting and creating an incoming connection, the Selector implicitly selects a SelectionKey with `ACCEPT` operation interest which is processed by the SelectorThread by calling `accept` on the socket channel. This creates a new socket channel and acknowledges the connection. Afterwards, the interest `READ` is registered in order to receive the node ID of the remote node. After selecting and dispatching the interest, the node ID is read by using the socket channel's read method.

At this point the socket channel is ready for sending and receiving data, but the ConnectionObject has yet to be created and pushed to the connection manager. This process is rather time consuming and might be blocking if an application thread creates a connection to the same node at the same time (connection duplication is discussed in Section V-2). Therefore, the SelectorThread creates a job for creating the connection and forwards it to the **ConnectionCreationHelper** thread. Additionally, the interest is set to `NO-OP` (0) to avoid receiving data before the connection setup is finished and the connection is attached to the SelectionKey.

The ConnectionCreationHelper polls the job queue periodically. There are two types of jobs: (1) a connection creation job and (2) a connection shutdown job. The latter is explained in Section V-3. When pulling a connection creation job, the ConnectionCreationHelper creates a new ConnectionObject (including the pipes, ORB, FC, etc.) and registers a READ interest with the new ConnectionObject attached. Furthermore, the PipeIn is marked as connected.

To be able to accept incoming connection requests, every node must open a `ServerSocketChannel`, bind it to a well-known port and register the `ACCEPT` interest. Furthermore, for selecting socket channels, a `Selector` has to be created and opened.

2) Connection Duplication: It is crucial to avoid connection duplication which occurs if two nodes create a connection to each other simultaneously. In this case, the nodes might use different connections to send and receive data which corrupts the message ordering and flow control. There are two approaches for resolving this problem: (1) detecting connection duplication during/after the connection establishment and (2) avoiding connection duplication by using two separate socket channels for sending and receiving.

Solution 1: **Detect and resolve connection duplication** by keeping one connection opened and closing the other one. Obviously, the other node must decide consistently which can be done by including the node IDs (e.g., always keep the connection created by the node with higher node ID). One downside of this approach is the complex connection shutdown. It must ensure that all data initially to be sent over the closing connection has been sent and received. Furthermore, message ordering cannot be guaranteed until the connection duplication situation is resolved.

Solution 2: **Avoid connection duplication** by using two socket channels per connection: one for sending and one for receiving (implemented in EthDXNet). Thus, simultaneous connection creation leads to **one** ConnectionObject with

opened PipeIn and PipeOut (one socket channel, each) whereas a single connection creation opens either the PipeOut (active) or PipeIn (passive). This approach requires additional memory for the second socket channel, Java.nio's Selector has more socket channels to manage and connection setup is required from both ends. The additional memory required for the second socket channel is negligible as the kernel socket buffers are configured to use a very small socket receive buffer for the outgoing socket channel and a very small socket send buffer for the incoming socket channel. The second TCP handshake (for connection creation, both sides need to open and connect a socket channel) is also not a problem as both socket channels can be created simultaneously and for a long running big data application connections among application instances are typically kept over the entire runtime. Finally, the overhead for Java.nio's Selector is difficult to measure but is certainly not the bottleneck taking into account the limitations of the underlying network latency and throughput. **Sending out-of-band (OOB) data** is possible by utilizing **the unused back-channel of every socket channel**. We use this for sending flow control data in EthDXNet (see Section VI).

3) Connection Shutdown: Connections are closed on three occasions: (1) if a write or read access to a socket channel failed, (2) if a new connection is to be created but the configurable connection limit is reached or (3) on node shutdown. In the first case, the SelectorThread directly shuts down the connection. In the second case, the application thread registers a `CLOSE` interest to let the SelectorThread close the connection asynchronously. On application shutdown, all connections are closed by one **Shutdown Hook** thread.

To shut down a connection, first, the outgoing and incoming socket channels are removed from the Selector by canceling the SelectionKeys representing a socket channel's registration. Then, the socket channels are closed by calling the socket channels' `close` method. At last, the connection is removed from the connection manager by creating a shutdown job handled by the ConnectionCreationHelper (case (1)) or directly removing it when returning to the connection management (cases (2) and (3)). The ConnectionCreationHelper also triggers a `ConnectionLostEvent`, which is dispatched to the application for further handling (e.g., node recovery).

When dismissing a connection (case (2)), directly shutting down a connection might lead to data loss. Therefore, the connection is closed gracefully by waiting for all outstanding data (in the connection's ORB) to be sent. Priorly, the connection is removed from connection management to prevent further filling of the ORB. Afterwards, a CLOSE interest is registered to close the socket channels asynchronously. The SelectorThread does not shut down the socket channels on first opportunity but postpones shutdown for at least two RTT timeouts to ensure all responses are received for still outstanding requests.

VI. Flow Control

DXNet provides a flow control on application layer to avoid overwhelming slower receivers (see Section III). EthDXNet uses the *Transmission Control Protocol* (TCP) which already implements a flow control mechanism on protocol layer. Still, DXNet's flow control is beneficial when using TCP. If the application on the receiver cannot read and process the data fast enough, the sender's TCP flow control window, the maximum amount of data to be sent before data receipt has to be acknowledged by the receiver, is reduced. The decision is based on the utilization of the corresponding kernel socket receive buffer. In DXNet, reading incoming data from kernel socket receive buffers is decoupled from processing the included messages, i.e., many incoming buffers could be stored in the IBQ to be processed by another thread. Thus, the kernel socket receive buffers' utilizations do not necessarily indicate the load on the receiver leading to delayed or imprecise decisions by TCP's flow control.

This section focuses on the implementation of the flow control in EthDXNet. Flow control data has to be sent with high priority to avoid unintentional slow-downs and fluctuations regarding throughout and latency. Sending flow control data in-band, i.e., with a special message appended to the data stream, is not an option because the delay would be too high. TCP offers the possibility to send **urgent data**, which is a single byte inlined in the data stream and sent as soon as possible. Furthermore, urgent data is always sent, even if the kernel socket receive buffer on the receiver is full. To distinguish urgent data from the current stream (urgent data can be at any position within a message as transfer is not message-aligned), a dedicated flag within the TCP header needs to be checked. This flag indicates if the first byte of the packet is urgent data. Unfortunately, Java.nio does not provide methods for handling incoming TCP urgent data.

We solve this problem **by using both unused back-channels** of every socket channel which are available because of the double-channel connection approach in EthDXNet. Thus, the incoming stream of the outgoing socket channel and the outgoing stream of the incoming socket channel of every connection are used **for sending/receiving flow control data**.

Sending flow control data: When receiving messages, a counter is incremented by the number of received bytes for every incoming buffer. If the counter exceeds a configurable threshold (e.g., 60% of the flow control window), a `WRITE_FC` interest is registered. This interest is applied, selected and dispatched like any other WRITE interest. But, instead of using the socket channel of the PipeOut, **the PipeIn is used to send the flow control data**. The flow control data consists of one byte containing the number of reached thresholds (typically 1). If the threshold is smaller than 50%, for example 30%, it is possible that between registering the WRITE_FC interest and actually sending the flow control data, the threshold has been exceeded again. For example, if the current counter is 70% of the windows size which is more than two thresholds of 30%. In this case 2 * 30% = 60% is confirmed by sending the value 2. After sending flow control data, the SelectionKey is reset to READ to enable receiving messages on this socket channel, again.

Receiving flow control data: To be able to receive flow control data, the socket channel of **the PipeOut must be readable** (register READ). If flow control data is available to be received, the socket channel is selected by the Selector and the SelectorThread reads the single byte from the socket channel of the PipeOut. When processing serialized messages on the sender, a counter is incremented. Application threads which want to send further messages if the counter reached

TABLE I. JAVA.NIO INTERESTS

Interest	Description
OP_READ	channel is ready to read incoming data
OP_WRITE	set if data is available to be sent
OP_CONNECT	set to open connection
OP_ACCEPT	a connect request arrived

TABLE II. ETHDXNET INTERESTS

Interest	Description (refers to attached connection)
CONNECT	set OP_CONNECT for outgoing channel
READ_FC	set OP_READ for outgoing channel
READ	set OP_READ for incoming channel
WRITE_FC	set OP_WRITE for incoming channel
WRITE	set OP_WRITE for outgoing channel
CLOSE	shutdown both socket channels

the limit (i.e., the flow control window is full) are blocked until message receipt is acknowledged by the receiver. The read flow control value is used to decremented the counter to re-enable sending messages. Usually, the limit is never reached as the flow control data is received before (if the threshold on the receiver is low enough).

In Section IV-A, we discussed the end-to-end situation of both nodes sending data to each other, but never reading (if the SelectionKey's operation stays at WRITE) causing a deadlock. This situation cannot occur with two socket channels per connection as reading and writing are handled independently. But, a similar situation is possible where two nodes send data to each other, but flow control data is not read for a while. This does not cause a deadlock but decreases performance. By setting the interest to READ | WRITE, flow control data is read from time to time ensuring a contiguous high throughput.

VII. EFFICIENT MANAGEMENT OF OPERATION INTERESTS

Operation interests are an important concept in Java.nio and are registered in the Selector to create and accept a new connection, to write data or to enable receiving data. The operation interests are complemented by the ConnectionObject (as an attachment) and the socket channel stored together in a SelectionKey. As soon as the socket channel is ready for any registered operation, the Selector adds the corresponding SelectionKey to a ready-set and wakes-up the SelectorThread waiting in the `select` method. If the SelectorThread is not waiting in the select method, the next select call will return immediately. The SelectorThread can then process all SelectionKeys.

A. Types of Operations Interests

The operation interests can be classified into two categories: **explicit operation interests and implicit operation interests**. Implicit operations are registered as presets after socket channel creation and after executing explicit operations. For example, a READ interest is registered for a socket channel if data is expected to arrive on this socket channel. The operation is then selected implicitly by the Selector whenever data is available to be received. Another example is the ServerSocketChannel which implicitly accepts new incoming connection requests if the ACCEPT interest has been registered before. Explicit operations are single operations which need to be triggered explicitly by the application. For example, when the application wants to send a message, the application thread has to register a WRITE interest. When the socket channel is ready, the data is sent and the socket channel is set to the preset (in our case READ). It is not forbidden by Java.nio to keep explicit operations registered. But, as a consequence the operations are always selected (every time select is called) which increases CPU load and latency. Therefore, in EthDXNet, every explicit operation is finished by registering an implicit operation.

The set of Java.nio operation interests is extended by EthDXNet to support flow control and to enable closing connections asynchronously. Table I shows all interests specified by Java.nio and Table II lists all interests used in EthDXNet. The interests READ, WRITE and CONNECT are directly mapped onto OP_READ, OP_WRITE and OP_CONNECT. OP_ACCEPT is registered and selected by the Selector and must not be registered explicitly. READ_FC and WRITE_FC are used to register OP_READ and OP_WRITE interests for the back-channel used by the flow control. The interest CLOSE does not have a counterpart because the method `close` can be called explicitly on the socket channel.

B. Interest Queue

None of the interests in Table II are registered directly to the Selector because only the SelectorThread is allowed to add and modify SelectionKeys. This is enforced by the Java.nio implementation which blocks all register calls when the SelectorThread is waiting in the `select` method. This obstructs the typical asynchronous application flow and can even result in a deadlock if the Selector does not have implicit operations to select. This problem can be avoided by always waking-up the SelectorThread before registering the operation interest and synchronizing the register and select calls. However, this workaround results in a rather high overhead and a complicated work flow. Instead, **we address this problem with an Interest Queue** (see Figure 4) and register all interests in one bulk operation executed by the SelectorThread before calling select. This approach provides several benefits while solving the above problem: first, **the application threads can return quickly** after putting the operation interest into the queue and even faster (without any locking) when the interest was already registered (which is likely under high load). Second, **the operation interests can be combined** and put in a semantic order (e.g., CONNECT before WRITE) before registering (a rather expensive method call). Finally, **the operation interest-set can be easily extended**, e.g., by a CLOSE operation interest to asynchronously shut down socket channels.

Figure 4 shows the Interest Queue consisting of a byte array storing the operation interests of all connections (left side in Figure 4) and an ArrayList of ConnectionObjects containing connections with new operation interests sorted by time of occurrence (right side in Figure 4).

The byte array has one entry per node ID allowing access time in O(1). The node ID range is limited to 2^{16} (allowing

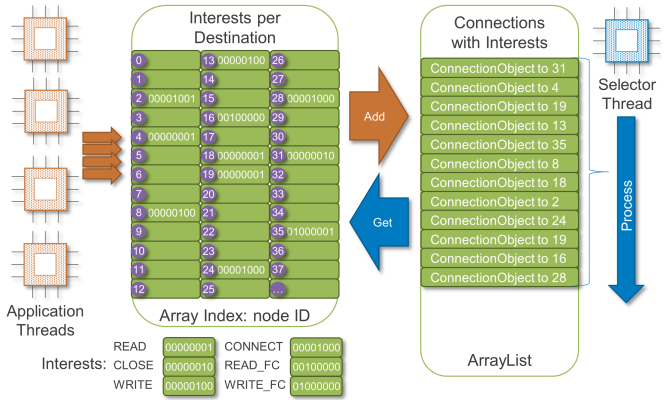

Figure 4. Interest Queue: the application threads add new interests to the Interest Queue. If interest was 0 before, the ConnectionObject is added to an ArrayList.

max. 65,536 nodes per application) which results in a fixed size of 64 KB for the byte array. An array entry is not zero if at least one operation interest was added for given connection to the associated node ID. Operation interests are combined with the bitwise or-operator to avoid overwriting any interest. By combining operation interests, the ordering of the interests for a single connection is lost. But, this is not a problem because a semantic ordering can be applied when processing them.

The ordering within the interests of one connection can be reconstructed but not the ordering across different connections. Therefore, whenever an interest is added to a non-zero entry of the byte array, the corresponding ConnectionObject is appended to an ArrayList. The order of operation interests is then ensured by processing the interest entries in the ArrayList in ascending order. The ArrayList also allows the SelectorThread to iterate only relevant entries and not all 2^{16}.

Processing operation interests: The processing is initiated either by the Selector implicitly waking up the SelectorThread if data is available to be read or an application thread explicitly waking up the SelectorThread if data is available to be sent. As waking-up the SelectorThread is a rather expensive operation (a synchronized native method call), it is important to call it if absolutely necessary, only. Therefore, the SelectorThread is woken-up after adding the first operation interest to the Interest Queue across all connections (the ArrayList is empty after processing the operation interests). If the SelectorThread is currently blocked in the select call, it returns immediately and can process the pending operation interests.

Listing 5 shows the basic processing flow of the SelectorThread. The first step in every iteration is to register all operation interests collected in the ArrayList of the Interest Queue. The SelectorThread gets the destination node ID from the ConnectionObject and the interests from the byte array. Operation interests are registered to the Selector in the following order:

1) CONNECT: register SelectionKey OP_CONNECT with given connection attached to an outgoing channel.
2) READ_FC: register SelectionKey OP_READ with given connection attached to an outgoing channel.
3) READ: register SelectionKey OP_READ with given connection attached to an incoming channel.

```
1        while (!closed) {
2        processInterests();
3
4        if (Selector.select() > 0) {
5        for (SelectionKey key :
             Selector.selectedKeys()) {
6        // Dispatch key
7        if (key.isValid()) {
8        if (key.isAcceptable()) {
9        accept();
10       } else if (key.isConnectable()) {
11       connect();
12       } else if (key.isReadable()) {
13       read();
14       } else if (key.isWritable()) {
15       write();
16       }
17       }
18       }
19       }
20       }
```

Figure 5. Workflow of SelectorThread

4) WRITE_FC: change SelectionKey of an incoming channel to OP_WRITE if it is not OP_READ | OP_WRITE.
5) WRITE: change SelectionKey of an outgoing channel to OP_WRITE if it is not OP_READ | OP_WRITE.
6) CLOSE: keep interest in queue for delay or close connection (see Section V-3).

The order is based on following rules: (1) a connection must be connected before sending/receiving data, (2) setting the preset READ is done after connection creation, only, (3) all READ and WRITE accesses must be finished before shutting down the connection and (4) the flow control operations have a higher priority than normal READ and WRITE operations. Furthermore, re-opening a connection cannot be done before the connection is closed and closing a connection is only possible if the connection has been connected before. Therefore it is not possible to register CONNECT and CLOSE together.

Finally, the processing of registered operation interests includes reseting the operation interest in the byte array and removing the ConnectionObject from the ArrayList.

VIII. Evaluation

We evaluated EthDXNet using up to 65 virtual machines (64 running the benchmark and one for deployment) connected with 5 GBit/s Ethernet in Microsoft's Azure cloud in Germany Central. The virtual machines are Standard_DS13_v2 which are memory optimized servers with 8 cores (Intel Xeon E5-2673), 56 GB RAM and a 10 GBit/s Ethernet connectivity, which is limited by SLAs to 5 GBit/s. In order to manage the servers, we created two identical scale-sets (as one scale-set is limited to 40 VMs) based on a custom Ubuntu 14.04 image with 4.4.0-59 kernel and Java 8.

We use a set of micro benchmarks for the evaluation sending messages or requests of variable size with a configurable number of application threads. All throughput measurements refer to the payload size which is considerably smaller than the full message size, e.g., a 64-byte payload results in 115 bytes to

TABLE III. ADDITIONAL PARAMETERS

Parameter	Value
ORB Size	4 MB
Flow Control Windows Size	2 MB
Flow Control Threshold	0.6
net.core.rmem_max	4 MB
net.core.wmem_max	4 MB

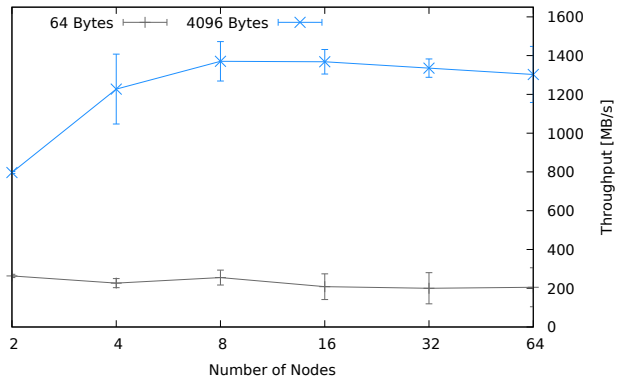

Figure 6. Message Payload Throughput per Node. 1 Application Thread, 2 Message Handler Threads

be sent on IP layer. Additionally, all runs with DXNet's benchmark are **full-duplex** showing the aggregated performance for concurrently sending and receiving messages/requests.

A. *Message Throughput*

First, we measured the asynchronous message throughput with an increasing number of nodes in an all-to-all test with message payloads of 64 and 4096 bytes. For instance, when running the benchmark with 32 nodes each node sends 25,000,000 64-byte messages to all 31 other nodes and therefore each node has to send and receive 775,000,000 messages in total. Additional network parameters can be found in Table III.

Figure 6 shows the average payload throughput for single nodes and Figure 7 the aggregated throughput of all nodes.

For 64-byte messages, the payload throughput is between 200 and 260 MB/s for all node numbers, showing a minimal decrease from 2 to 16 nodes. With 4096-byte messages the throughput improves with up to 8 nodes peaking at 1370 MB/s full-duplex bandwidth (5.5 GBit/s uni-directional). With 64 nodes the throughput is still above 5 GBit/s resulting in an aggregated throughput of 83,376 MB/s. The minor decline in both experiments can be explained by an uneven deployment of our network benchmark causing the last nodes starting and finishing a few seconds later. The end-to-end throughput between two nodes seems to be bound at around 3.2 GBit/s in the Microsoft Azure cloud as tests with iperf showed, too.

The benchmarks show that DXNet, as well as EthDXNet scale very well for asynchronous messages under high loads.

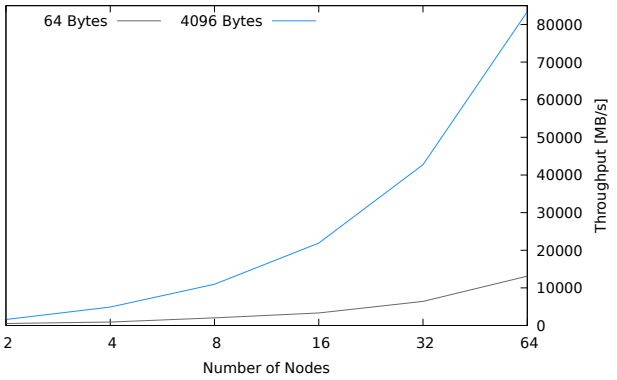

Figure 7. Aggregated Message Payload Throughput. 1 Application Thread, 2 Message Handler Threads

B. *Request-Response Latency*

The next benchmarks are used to evaluate request-response latency by measuring the **Round Trip Time** (RTT) which includes sending a request, receiving the request, sending the corresponding response and receiving the response. Figure 8 shows the RTTs for an all-to-all scenario with 2 to 64 nodes and 1, 16 and 100 application threads. Furthermore, all-to-all tests with `ping` are included to show network latency limitations.

The latency of the Azure Ethernet network is relatively high with a minimum of 352 µs measured with DXNet and one application thread (Figure 8). A test with up to 4032 ping processes shows that the average latency of the network is even higher (> 500 µs). In DXNet, own requests are combined with responses (and other requests if more than one application thread is used). This reduces the average latency for requests. Additionally, the ping baseline shows an increased latency for more than 32 nodes, by using one scale-set for the first 32 nodes and another one for the last 32 nodes. Different scale-sets are most likely separated by additional switches which increases the latency for communication between scale-sets.

EthDXNet is consistently under the ping baseline demonstrating the low overhead and high scalability of EthDXNet (and DXNet) when using one application thread. With 16 application threads, the latency is slightly higher and on the same level as the baseline, but the throughput is more than 10 times higher as well (in comparison to DXNet with one application thread). Furthermore, both lines have the same bend from 32 to 64 nodes as the baseline.

With 100 application threads per node (up to 6,400 in total), the latency increases noticeably, as expected, because the CPU is highly overprovisioned. In this situation the latency between writing a message into the ORB and sending it increases dramatically with more open socket channels. Furthermore, requests can be aggregated more efficiently in the ORBs with less open connections masking the overhead with a few nodes.

The latency experiments show that EthDXNet scales up to 64 nodes without impairing latency. With a very high number of application threads (relative to the available cores) the latency increases but is still good.

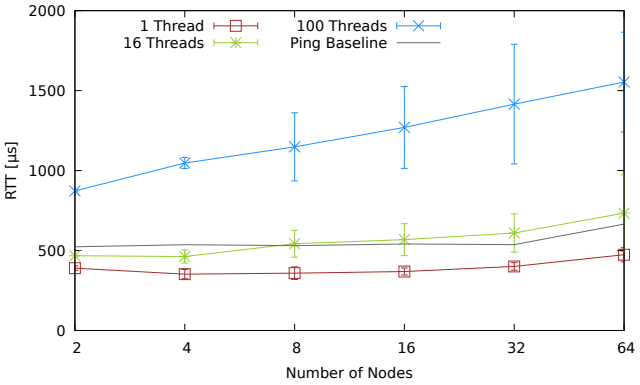

Figure 8. Average Request-Response Latency. 1 to 100 Application Threads, 2 Message Handler Threads

IX. Conclusions

Big data applications, as well as large-scale interactive applications are often implemented in Java and typically executed on many nodes in a cloud data center. Efficient network communication is very important for these application domains.

In this paper, we described our practical experiences in designing a transport implementation, EthDXNet, based on Java.nio, integrated into DXNet. EthDXNet provides a double-channel based automatic connection approach using back-channels for sending flow control data and an efficient operation interest handling which is important to achieve low-latency message handling with Java.nio's Selector.

Evaluation with micro benchmarks in the Microsoft Azure cloud shows the scalability of EthDXNet (together with DXNet) achieving an aggregated throughput of more than 83 GByte/s with 64 nodes connected with 5 GBit/s Ethernet (10 GBit/s Ethernet limited by SLAs). Request-response latency is almost constant for an increasing number of nodes as long as the CPU is not overloaded. Future work includes experiments on larger scales with application traces.

References

[1] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, "One trillion edges: Graph processing at facebook-scale," *Proc. VLDB Endow.*, vol. 8, pp. 1804–1815, Aug. 2015.

[2] S. Ekanayake, S. Kamburugamuve, and G. C. Fox, "Spidal java: High performance data analytics with java and mpi on large multicore hpc clusters," in *Proceedings of the 24th High Performance Computing Symposium*, 2016, pp. 3:1–3:8.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[4] S. Microsystems, "Java remote method invocation specification," https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html, accessed: 2018-03-14.

[5] Oracle, "Package java.net," https://docs.oracle.com/javase/8/docs/api/java/net/package-summary.html, accessed: 2018-03-14.

[6] S. Mintchev, "Writing programs in javampi," School of Computer Science, University of Westminster, Tech. Rep. MAN-CSPE-02, Oct. 1997.

[7] K. Beineke, S. Nothaas, and M. Schoettner, "Efficient messaging for java applications running in data centers," Feb. 2018, preprint on webpage at https://cs.hhu.de/en/research-groups/operating-systems/publications.html.

[8] S. P. Ahuja and R. Quintao, "Performance evaluation of java rmi: A distributed object architecture for internet based applications," in *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '00, 2000, pp. 565–569.

[9] M. Philippsen, B. Haumacher, and C. Nester, "More efficient serialization and rmi for java," *Concurrency: Practice and Experience*, vol. 12, pp. 495–518, 2000.

[10] R. Latham, R. Ross, and R. Thakur, "Can mpi be used for persistent parallel services?" in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Berlin Heidelberg, 2006, pp. 275–284.

[11] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using mpi in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, ser. EuroMPI '13, 2013, pp. 43–48.

[12] Oracle, "Java i/o, nio, and nio.2," https://docs.oracle.com/javase/8/docs/technotes/guides/io/index.html, accessed: 2018-03-14.

[13] W. Pugh and J. Spacco, *MPJava: High-Performance Message Passing in Java Using Java.nio*. Springer Berlin Heidelberg, 2004, vol. 16.

[14] R. Hitchens, *Java NIO*. Sebastopol, CA, USA: O'Reilly Media, 2009.

[15] G. L. Taboada, J. Touriño, and R. Doallo, "Java fast sockets: Enabling high-speed java communications on high performance clusters," *Comput. Commun.*, vol. 31, pp. 4049–4059, Nov. 2008.

[16] K. Beineke, S. Nothaas, and M. Schoettner, "Dxnet project on github," https://github.com/hhu-bsinfo/dxnet, accessed: 2018-03-14.

Chapter 4.

High Throughput Log-based Replication for Many Small In-memory Object

This chapter summarizes the contributions and includes a copy of our paper [15]. A short version of the contributions presented in [15] were transferred to a poster with a peer-reviewed two-page abstract [16].

Paper: Kevin Beineke, Stefan Nothaas and Michael Schöttner. "High Throughput Log-Based Replication for Many Small In-Memory Objects". In: IEEE 22nd International Conference on Parallel and Distributed Systems. Dec. 2016, pp. 535–544

Poster: Kevin Beineke, Stefan Nothaas and Michael Schöttner. "High Throughput Log-Based Replication for Many Small In-Memory Objects". In 2016 IEEE International Conference on Cluster Computing (CLUSTER). Sept. 2016, pp. 160-161

4.1. Paper Summary

This paper describes and evaluates the logging architecture of DXRAM whose basic concepts were firstly introduced in [10] (can be found in the Appendix 8.2).

DXRAM stores all data objects in RAM providing low-latency access to billions of small data objects. As RAM is typically more expensive than disk space, replicating to other server's main memory to enable fault-tolerance comes at a high price. Additionally, if many servers fail simultaneously, data might be lost as no persistent copy exists. Therefore, we transparently store all replicas in logs on remote disks, preferably SSDs. In this publication, we propose a two-level logging approach which combines high throughput with fast persistence while being memory efficient. Furthermore, a novel backup-side version management and the concurrent reorganization is detailed in this publication.

The evaluation shows that DXRAM outperforms other state-of-the-art in-memory key-value stores like RAMCloud, Redis and Aerospike regarding the write throughput and memory overhead of the backup components and also regarding scalability of the number of servers.

4.2. Importance and Impact on Thesis

Replication is mandatory to enable fault-tolerance which is the main topic of this thesis. In this paper, we describe the replication mechanism of DXRAM which logs all replicas to logs on remote servers. All proposed concepts are optimized for DXRAM's primary application domains, i.e., the logging and version management was designed to handle billions of small data objects and access patterns like Zipf and random distributions. Furthermore, the concepts were developed to be least interfering in fault-free execution but also to enable a very fast recovery (see Chapter 5). The replica placement is introduced in this publication and further discussed in Chapter 5 and 6. The log selection strategy for the reorganization is also continued in Chapter 6.

4.3. Personal Contribution

The basic logging architecture was conceived by Dr. Florian Klein, Prof. Dr. Michael Schöttner and Kevin Beineke, the author of this thesis, in discussions prior to the doctoral studies. This includes the basic concept of the two-level logging and first thoughts regarding the reorganization and recovery which were published in [10]. While the idea of the two-level logging was transferred to this paper, major alterations to the backup distribution, reorganization and recovery were applied by Kevin Beineke. Additionally, the novel backup-side version management was introduced by Kevin Beineke in order to efficiently identify outdated log entries during reorganization and recovery.

The implementation was based on a master thesis [51] by Yunus Kaplan. The thesis aimed at implementing the first, basic concept described above. Because of the fundamentally refined architecture and optimization reasons, most of the implementation was replaced by Kevin Beineke. All depicted concepts were implemented in DXRAM and evaluated by the author of this thesis.

Kevin Beineke structured and wrote most of the paper, including all figures. The local data management of DXRAM (section II.B.) was outlined by Stefan Nothaas, who also reviewed and proof-read the paper and participated in discussions. Prof. Dr. Michael Schöttner reviewed this paper as well, advanced the structure and verbalization, and helped improving comprehensibility.

High Throughput Log-based Replication for Many Small In-memory Objects

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—Online graph analytics and large-scale interactive applications such as social media networks require low-latency data access to billions of small data objects. These applications have mostly irregular access patterns making caching insufficient. Hence, more and more distributed in-memory systems are proposed keeping all data always in memory. These in-memory systems are typically not optimized for the sheer amount of small data objects, which demands new concepts regarding the local and global data management and also the fault-tolerance mechanisms required to mask node failures and power outages. In this paper we propose a novel two-level logging architecture with backup-side version control enabling parallel recovery of in-memory objects after node failures. The presented fault-tolerance approach provides high throughput and minimal memory overhead when working with many small objects. We also present a highly concurrent log cleaning approach to keep logs compact. All proposed concepts have been implemented within the DXRAM system and have been evaluated using two benchmarks: The Yahoo! Cloud Serving Benchmark and RAMCloud's Log Cleaner benchmark. The experiments show that our proposed approach has less memory overhead and outperforms state-of-the-art in-memory systems for the target application domains, including RAMCloud, Redis, and Aerospike.**

***Keywords*—Cloud computing; Data centers; Reliability; Remote replication; Main memory; Secondary storage; Flash memory; Buffering; B-trees; Graph-based database models**

Large-scale interactive applications and online graph processing often work with huge amounts of very small data objects. Facebook, for example, stores billions of small data objects with most of them smaller than 64 byte [1], building an enormous graph. Other graph examples are brain simulations with billions of neurons and thousands of connections each [2] or search engines for billions of indexed web pages [3].

These applications all have a need for low-latency data-access to process online analytics or interactive queries. Storage systems such as traditional databases or in-memory storages often fail to handle small data objects efficiently and introduce a considerable large meta-data overhead on a per object basis. Therefore, it is often recommended to aggregate graph vertices and edges which impacts latency and is burdening the developer. By holding all single objects always in RAM, the latency is reduced dramatically, but storing the objects as compact as possible becomes more important because RAM is more expensive than flash or disk storage.

In-memory caches like Memcached [4] or Gemfire [5] provide low latency but not fault-tolerance and thus the programmer is burdened to keep caches and back-end storage synchronized which is challenging and error prone. Furthermore, because of the often irregular data access patterns to reduce costly cache misses, the majority of objects have to be cached. Facebook, for instance, has used up to 1,000 memcached servers to store around 75% of all data in RAM [6]. Still, for every write access the back-end storage needs to be updated impairing latency.

The excessive use of volatile memory, either as a cache or primary storage, requires sophisticated fault-tolerance mechanisms in order to avoid data loss in case of power outages and node failures. It is important to minimize the impact on throughput during fault-free execution while allowing to quickly recover failed nodes. Distributed in-memory storage systems like Redis [7], Aerospike [8], RAMCloud [9] and DXRAM [10] address these problems by keeping all data always in memory combined with a transparent logging mechanisms on secondary storage to prevent data loss in case of errors. The architectures of state-of-the-art distributed in-memory systems are described in section I.

DXRAM is a distributed in-memory storage designed to efficiently support many small data objects. The latter covers a minimal meta-data overhead, scalability regarding number of storage nodes and high throughput for client requests. The system is designed to run within a single data center (currently over Gigabit Ethernet, Infiniband planned). The memory management and the global meta-data management are described briefly in this paper (for more information refer to [10]). However, the focus of this publication is on the logging mechanisms.

The contributions of this paper are:

- a novel two-stage logging approach enabling fast recovery and providing high throughput while being memory efficient
- a backup-side version control which was designed with a very low memory footprint and allows a high object creation and update throughput
- a highly concurrent log cleaning concept also designed for handling many small data objects
- a backup performance evaluation and comparison with state-of-the-art distributed in-memory systems

The evaluation with two benchmarks and comparisons with state-of-the-art in-memory systems on a cluster show that the proposed logging concepts are fast and efficient and allow a

high throughput. The results show that using DXRAM with logging and reorganization enabled clearly outperforms similar systems for a broad application domain.

The structure of this paper is as follows. Related work is discussed in section I, followed by a brief overview of relevant background information about DXRAM in section II. In section III and IV the backup-side logging architecture is presented first, followed by the log reorganization approach described in section V. Conclusions and an outlook on future work are found in the last section VII.

I. Related Work

Numerous distributed in-memory systems have been proposed to provide low-latency data access for online queries and analytics for various graph applications. These systems often need to aggregate many nodes to provide enough RAM capacity for the exploding data volumes which in turn results in a high probability of node failures. The latter includes soft- and hardware failures as well as power outages which need to be addressed by replication mechanisms and logging concepts storing data on secondary storage. Because of space constraints, we can only discuss the most relevant work.

RAMCloud is an in-memory system, sharing several objectives with DXRAM while having a different architecture, providing a table-based in-memory storage to keep all data always in memory. However, the table-based data model of RAMCloud is designed for larger objects and suffers from a comparable large overhead for small data objects [10]. It uses a distributed hash table, maintained by a central coordinator, to map 64-bit global IDs to nodes which can also be cached by clients. DXRAM on the other hand uses a superpeer overlay with a more space-efficient range-based meta-data management. For persistence and fault tolerance it implements a log-based replication of data on remote nodes' disks [9]. In contrast to other in-memory systems, RAMCloud organizes in-memory data also as a log which is scattered for replication purposes across many nodes' disks in a master slave coupling [11]. Scattering the state of one node's log on many backup nodes allows a fast recovery of 32 GB of data and more. Obviously, logging throughput depends on the I/O bandwidth of disks as well as on the available network bandwidth and CPU resources for data processing. RAMCloud uses a centralized log-reorganization approach executed on the in-memory log of the server which resends re-organized segments (8 MB size) of the log over the network to backup nodes. As a result, remaining valid objects will be re-replicated over the network after every reorganization iteration to clean-up the persistent logs on remote nodes. This approach relieves remote disks but at the same time burdens the master and the network. DXRAM uses an orthogonal approach by doing the reorganization of logs on backup nodes avoiding network traffic for reorganization. Furthermore, DXRAM does not organize the in-memory storage as a log but uses updates in-place. Finally, RAMCloud is written in C++ and provides client bindings for C, C++, Java and Python [12] whereas DXRAM is written in Java.

Aerospike is a distributed database platform providing consistency, reliability, self-management and high performance clustering [8]. Aerospike uses Paxos consensus for node joining and failing and balances the load with migrations. In comparison, DXRAM also offers a migration mechanism for load balancing. The object lookup is provided by a distributed hash table in Aerospike. Like DXRAM, Aerospike is optimized for TCP/IP. Additionally, Aerospike enables different storage modes for every namespace. For instance, all data can be stored on SSD with indexes in RAM or all data can be stored in RAM and optionally on SSD with a configurable replication factor. As Aerospike is a commercial product, not many implementation details are published except that it internally writes all data into logs stored in larger bins optimized for flash memory. The basic server code of Aerospike is written in C and available clients include bindings for C, C#, Java, Go, Python, Perl and many more.

Redis is another distributed in-memory system which can be used as an in-memory database or as a cache [7]. Redis provides a master-slave asynchronous replication and different on-disk persistence modes. To replicate in-memory objects, exact copies of masters, called slaves, are filled with all objects asynchronously. To overcome power outages and node failures, snapshotting and append-only logging with periodical rewriting can be used. However, to replicate on disk the node must also be replicated in RAM which increases the total amount of RAM needed drastically. This is an expensive approach and very different from the one of DXRAM where remote replicas are stored on SSD only. Obviously, Redis has no problems with I/O bandwidth as it stores all data in RAM on slaves and can postpone flushing on disk as needed. Furthermore, reorganization is also quite radical compared to DXRAM as Redis just reads in a full log to compress it which is of course fast but introduces again a lot of RAM overhead. Redis is written in C and offers clients for many programming languages like C, C++, C#, Java and Go.

Apache Spark is a cluster computing framework which supports applications with working sets [13]. Data is held in a resilient distributed dataset (RDD), "a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost". Each RDD is a Scala object and can be created in four ways: from a file, by dividing an array, by transformation of an existing RDD and by changing the persistence of an existing RDD. By default a RDD is constructed every time it is accessed. If a RDD is used multiple times, it is possible to cache a RDD in memory. If there is not enough memory to hold a RDD partition, the system can swap it to HDFS or delete it and recompute it on demand. Typically, RDDs are large objects, containing large data sets. For example the graph extension GraphX holds a complete graph in a RDD. DXRAM, in contrast to Spark, is not a cache but a storage service specifically designed for many small data objects.

Log-structured File Systems are an important inspiration for the log-based replication of RAMCloud and DXRAM. A log is the preferred data structure for replication on disk as a log has a superior write throughput due to appending objects, only. But, a log requires a periodical reorganization to discard outdated or deleted objects in order to free space for further write accesses. In [14] Rosenblum and Ousterhout describe a file system which is based on a log. Furthermore, a cleaning policy is introduced which divides the log into segments and selects the segment with best cost-benefit ratio

for reorganization. DXRAM divides a log into segments as well. However, due to memory constrains the cost-benefit formula is limited to the age of a segment (more in section V).

Journaling is used in several file systems to reconstruct corruptions after a disk or system failure. A journal is a log that is filled with meta-data (and sometimes data) before committing to main file system. The advantage is an increased performance while writing to the log as appending to a log is faster than updating in-place but requires a second write access. The to be described two-level logging of DXRAM also uses an additional log to efficiently utilize an SSD. In contrast to journaling, we use this log only for small write accesses from many remote nodes to allow bulk writes without impeding persistence.

II. DXRAM Architecture Overview

DXRAM is a distributed in-memory system written in Java with a layered architecture which is open for additional services and data models beyond the key-value foundation of the DXRAM Core [10]. In DXRAM an in-memory data object is called a *chunk*. Objects that are stored in a log, on the other hand, are referred to as *log entry*. The term *object* is used further on when the location (log or memory) is unspecified or irrelevant.

A. Global Meta-Data Management

In DXRAM, every node is either a peer or a superpeer. Peers store chunks, may run computations and exchange data directly with other peers, and also serve client requests when DXRAM is used as a back-end storage. Superpeers store global meta-data like the locations of chunks, implement a monitoring facility, detect failures and coordinate the recovery of failed nodes, and also provide a naming service. The superpeers are arranged in a Chord-like overlay [15] adapted to the conditions in a data center (e.g. every superpeer has a global view as maintaining it is unpretentious with far less churn [16]). Moreover, every peer is assigned to one superpeer which is responsible for meta-data management and recovery coordination of its associated peers.

Every chunk in DXRAM has a 64-bit globally unique chunk ID (CID). This ID consists of two separate parts: A 16-bit node ID of the chunk creator and a 48-bit locally unique sequential number. Thereby, 65,536 nodes with around 280 trillion chunks per node are addressable. With the creator's node ID being part of a CID, every chunk's initial location is known a-priori. But, the location of a chunk may change over time in case of load balancing decisions or when a node fails permanently. Superpeers use a modified B-tree [17] allowing a space efficient and fast node lookup while supporting chunk migrations. Space efficiency is achieved by a per-node sequential ID generation and ID re-usage in case of chunk removals allowing to manage chunk locations using CID ranges with one entry for a set of chunks. In turn, a chunk location lookup will reply with a range of CIDs, not only a single location. This helps reducing the number of location lookup requests. For caching of lookup locations on peers, a similar tree is used further reducing network load for lookups. More details about this data structure can bee found in [10].

B. Memory Management

The sequential order of CIDs (as described in section II-A) allows us to use compact paging-like address translation tables on peers with a constant lookup complexity. Although, this table structure has similarities with well known operating systems' paging tables we apply it in a different manner. On each DXRAM peer we use the lower part (LID) of the CID as a key to lookup the virtual memory address of the stored chunk data. The LID is split into multiple parts (e.g. 4 parts of 12 bit each) representing the distinct levels of the paging hierarchy. This allows us to allocate and free page tables on demand reducing the overall memory consumption of the local meta-data management. Complemented with an additional level indexed by node ID storing of migrated chunks is possible as well. DXRAM uses a tailored memory allocator with very low footprint working on a large pre-reserved memory block [10]. For performance reasons, all memory operations are implemented using the Java Unsafe class.

III. Logging Architecture

In this section we describe the logging architecture of DXRAM. Regarding the logging backup system of DXRAM we distinguish two different roles: *Masters* are DXRAM peers, store chunks (like described in section II) and replicate them on *backup peers*. A backup peer might also be a master and vice versa.

A. Backup Data Structure

Replicating multi-billion small data objects in RAM is too expensive and does not allow to mask power outages. Therefore the backup data structures of DXRAM are designed to maximize throughput of SSDs devices:

1) SSDs write at least one flash page (default: 4 KB), pages are clustered to be accessed in parallel.
2) SSDs cannot overwrite a single flash page, but delete a block (64 to 128 pages) and write on another.
3) It is faster to write sequentially than randomly on SSDs because of the clustering.

With those characteristics in mind, it is apparent that updating or deleting objects of a few dozen bytes in place would slow down SSD throughput dramatically. Furthermore, storing SSD object locations in RAM would result in an additional meta-data overhead of at least 16 Byte (8-byte address on SSD and 8-byte CID) per object or 48 Byte if every object is replicated three times. For example, for a billion of 32-byte objects this would result in around 48 GB of additional RAM consumption or around 150% respectively for keeping track of object locations on SSD. Storing these location mappings on SSD, instead, either partially or fully would heavily burden SSD throughput due to a write access requiring an additional read access.

Therefore, we decided to use a logging approach for storing replicas on persistent memory. This solution does not require knowledge about the location of an object and ensures maximal write throughput as log entries are always appended resulting in sequential writing during normal operation. The log is only read during reorganization and recovery also benefiting from the sequential arrangement.

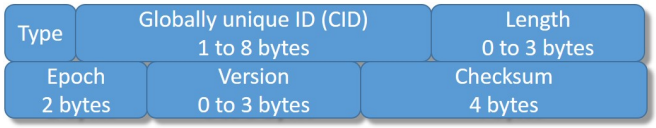

Figure 1: Log Entry Header. *The size of the header is dynamic, as the CID, length and version are only as large as necessary. Type: Depending on the location and whether the chunk was migrated or not. The purpose of the epoch field is described in section IV.*

B. Self-descriptive Log Entries

In DXRAM, all log entries are self-descriptive. This accelerates the recovery process as all necessary meta-data is always stored with the data itself. Furthermore, we can avoid an additional costly data structure for meta-data lookup. To reduce memory consumption on SSD the log entry headers have dynamically growing fields (Figure 1). This is very important for small data objects as header size can be reduced by up to 70 %. This also increases the object processing performance by reducing the data volume for I/O operations. For fault-tolerance reasons the log entry header also include a CRC32 checksum that is checked during recovery process to guarantee data integrity.

C. Backup Zones

DXRAM splits every master's data into backup zones of a configurable size (e.g. 256 MB, good value for fast recovery) and scatters the set of all backup zones to many backup peers in order to allow fast parallel recovery. When a master fails the associated superpeer controls the recovery process by interacting with backup peers to recover backup zones in parallel either in their own memory if enough space is available or on a fresh master. Consistency problems among the different backup zone replicas are avoided by a sequential backup order. The primary backup peer receives replica updates of a backup zone always first, then the secondary, and so on; and the primary is also prompted first to recover the backup zone if the corresponding master failed (Figure 2).

D. Two-Level Log Organization

In contrast to RAMCloud, we store each backup zone in one separate log on every assigned backup peer. Those logs are called *secondary logs* and are the final destination for every replica and the only data structure used to recover data from. By sorting backups per node we can speed-up the recovery process by avoiding to analyze a single log with billions of entries mixed from several masters.

The two-level log organization also ensures that infrequent written secondary logs do not thwart highly burdened secondary logs by writing small data to SSD and thus utilizing the SSD inefficiently. At the same time, incoming objects are quickly stored on SSD to sustain power outages.

First, every object received for backup is written to a ring buffer, called *primary buffer*, to bundle small request. This buffer is divided into buckets which allows concurrently writing into the buffer while it is partly flushed to SSD. During the flushing process, which is triggered periodically or if a threshold is reached, the bucket is sorted by backup zones

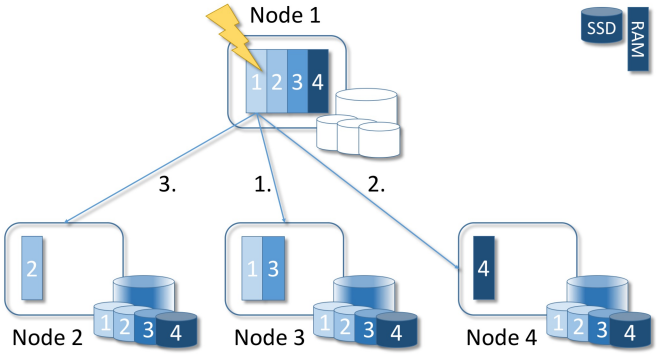

Figure 2: Backup Zones *The master's data (node 1) is scattered to three backup peers (nodes 2, 3 and 4). After failure of node 1, the backup peers recover all backup zones respecting the backup order (e.g. node 3 recovers the backup zone 1 from node 1 as it is the primary backup peer for backup zone 1).*

to form larger piles of data in order to allow bulk writes on SSD. If one of those piles is larger than a predefined threshold (e.g. 32 flash pages of the SSD), it is written directly to the corresponding secondary log.

In addition to the secondary logs, there is **one** *primary log* for temporarily storing smaller piles of **all** backup zones to guarantee fast persistence without decreasing SSD throughput. The smaller piles are also buffered in RAM separately, in so called *secondary log buffers*, for every secondary log and will eventually be written to the corresponding secondary log when aggregated to a larger pile (Figure 3). For example, if the primary buffer contains 256 KB of data, 128 KB from backup zone 1 and 128 KB evenly split over 64 additional backup zones (2 KB for each backup zone), then 128 KB will be written directly to secondary log 1 and the other 128 KB to the primary log. Additionally, the secondary log buffers of the 64 other backup zones are filled with 2 KB each. If, by appending the data, the threshold of one secondary log buffer is reached, it will be flushed to the corresponding secondary log. Obviously, with this approach some objects will be written twice to SSD but this is outweighed by utilizing the SSD more efficiently. Waiting individually for every secondary log until the threshold is reached without writing to primary log, on the other hand, is no option as the data is prone to get lost in case of a power outage.

The proposed logging architecture can also efficiently handle unbalanced access patterns. In social media networks, for example, a zipfian access pattern is expected. This means that the second most popular object is accessed half as often as the most popular one resulting in a pattern where many objects are rather seldom accessed but some very often. Transferred to the backup zones, this results in some backup zones being flooded with updates and lots of backup zones getting only a few updates. With an unbalanced approach, either many small write accesses would slow down the SSD (if every object is directly written to SSD) or buffering would make infrequently written logs vulnerable for data loss (if objects are buffered until a threshold is breached). The two-level log organization handles both situations effectively.

After a recovery process is initiated on a backup peer the primary buffer and the specific secondary log buffer must

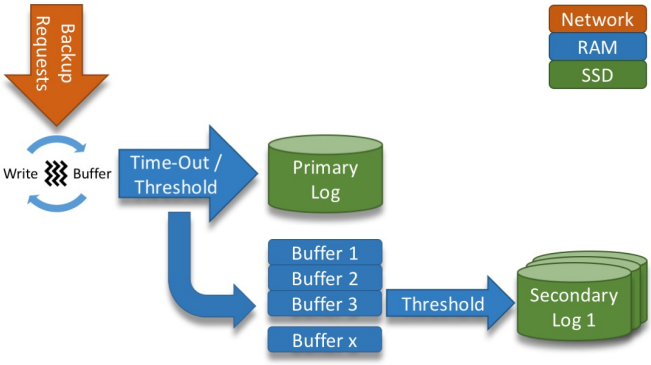

Figure 3: Logging architecture. *Every object is buffered first. Depending on the amount of data per backup zone, the objects are either directly written to the specific secondary log or to primary log and to secondary log once there is enough data.*

be flushed in order to have all relevant log entries in the corresponding secondary log. As the primary buffer is flushed frequently and the secondary log buffers have a rather small limited size, e.g. 128 KB, this is a fast operation. The primary log is not involved in a normal recovery process as after flushing of the secondary log buffer all log entries written to primary log are also stored in the specific secondary log. If the primary log is full, a flushing of all secondary log buffers will be sufficient to clear the primary log.

IV. Backup-side Version Control

Masters do not store version information in RAM, as there is no need when using updates in-place, but only backup peers on SSD. All the logic for version control and reorganization is outsourced to backup peers which only receive raw updates, deletes and creates including CID and backup zone ID. The backup-side version control, described in this section, is the foundation for recovery and log reorganization.

A. Order-Preserving Network Message Processing

In order to enable a backup-side version control, one must guarantee that the ordering of backup requests from one master does not change until the data is written to SSD by the backup peer. This does not require a global sequence but only a FIFO guarantee between the application layers of master and backup peer. On the network layer we are relying on the ordering of TCP and the message processing steps implemented in DXRAM are connected by synchronized queues allowing a highly parallel execution without sacrificing ordering. Message processing throughput is maximized by enforcing ordering only for exclusive messages whereas normal messages are handled concurrently by a thread pool. Every message type can be declared exclusive by the programmer if sequential processing is needed. Backup requests are exclusive messages thus subject to enforced ordering until saved in the primary buffer. The writing to SSD is then executed asynchronously.

B. Version Manager

Every log entry needs version information allowing to detect outdated versions. This is important for the recovery and also for the reorganization of logs which will be discussed in section V. A naïve solution would be to manage every object's version in RAM on backup peers. Unfortunately, this approach consumes too much memory, e.g. at least 12 bytes (8-byte CID and 4-byte version) for every object stored in logs easily summing up to many GB in RAM which is not affordable. Storing version information on SSD, only, is also not practical because of performance reasons as this would require reads for each log write. Caching recent versions in memory could possibly help for some access patterns but for the targeted application domain would either cause many read accesses for cache misses or occupy a lot of memory. Instead, we propose a version manager which runs on every backup peer and utilizes one *version buffer* per secondary log. The version buffer holds recent versions for this secondary log in RAM until it is flushed to SSD. In contrary to a simple cache solution, DXRAM's version manager avoids loading missing entries from secondary storage by distinguishing time spans, called *epochs*, which serve as an extension of a plain version number.

At the beginning of an epoch, the version buffer is empty. If a backup arrives within this epoch, its CID will be added to the corresponding version buffer with version number 0. Another backup for the same object within this epoch will increment the version number to 1, the next to 2 and so on. When the version buffer is flushed to SSD, all version information is complemented by the current epoch, together creating a unique version. In the next epoch the version buffer is empty again.

Two unique versions are in chronicle relation if:

$$[\,\text{Version x, Epoch i}\,] < [\,\text{Version y, Epoch j}\,],$$
$$\text{where } (i < j) \text{ or } (i = j \text{ and } x < y)$$

With the proposed approach, we can avoid reading version information from SSD when appending a new log entry. Still, unambiguous version are assigned.

As described before, the time between two flushes is called an epoch. An epoch ends when the version buffer reaches a predefined threshold allowing to limit buffer size, e.g. 1 MB per log. Different logs can be in different epochs which makes it unlikely that all version buffers reach the threshold at the same time. Nevertheless, the *peak* memory usage for such a situation is still acceptable, e.g. 128 MB for 32 GB payload stored in 128 logs. In comparison, storing all versions always in RAM for the given scenario and 32-byte objects would permanently consume around 10 GB memory on each backup peer.

For the version buffer we use a hash table with linear probing providing fast access while having control over memory consumption. During flushing to SSD, a version buffer is compacted resulting in a sequence of (CID, epoch, version)-tuples with no ordering. This sequence is appended to a file on SSD, creating a log of unique versions for every single secondary log (Figure 4). We call it a *version log*. Over time, a version log contains several invalid entries which are tuples with outdated versions. To prevent a version log from continuously growing, it is compacted during reorganization (discussed in section V). The resulting version log includes every CID at most once, discarding all invalid entries.

Overflow: In the presented approach, we try to minimize the bit lengths for versions and epochs to reduce memory usage for the version buffer in RAM and the version log on

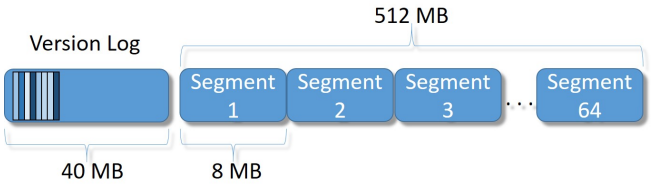

Figure 4: Complete secondary log. *A secondary log with version log for 256 MB backup zones and 64-byte objects.*

SSD. Thus, we can distinguish 2^{16} epochs and 2^{24} versions per object which requires overflow handling. For versions, an overflow can occur only for uncommon patterns like constantly counting up a shared variable but this can be easily handled by subsequently initiating flushing the version buffer. An epoch overflow on the other hand is more difficult to address.

We call a complete epoch iteration (0, 1, 2, ..., 2^{15}-1, 2^{15}) an *eon*. At any time, two eons are distinguishable by using the highest bit of the epoch number. The duration of an eon depends on the update rate of a secondary log, e.g. for 10,000 updates/s the eon switches every 3 days. In general, an epoch overflow poses no problem as for validity check an (epoch, version)-tuple of a log entry is only checked for equality against the current tuple in version manager.

There is however another rare case that requires an additional overflow handling: If an object is written very seldom, it cannot be ruled out that there are two log entries with the same version and epoch but separated by an entire eon. Thereby, it is impossible to identify the invalid log entry of the two without considering the eon. By comparing the eons of two entries with the current eon of the secondary log the log entries are stored in one can detect the more recent entry. But, this does not work if two entries are separated by more than one eon (version and epoch still the same). Thus, during one eon we transfer every valid log entry that was created in the former eon to the new eon by applying the following validity check:

```
Version currentVersion
  = VersionManager.getVersion(logEntry.CID);
if(logEntry.epoch != currentVersion.epoch ||
  logEntry.version != currentVersion.version){
  // Either version, epoch or eon is unequal
  // -> there is a more recent version in log
  remove(logEntry);
} else {
  if(currentVersion.eon != log.currentEon){
    // Checked log entry is the most recent,
    // but was created in last eon
    // -> transfer to current eon
    logEntry.eon = log.currentEon;
    // -> update VersionManager as well
    currentVersion.eon = log.currentEon;
  }
}
```

This fast validity check is executed asynchronously by a dedicated thread during reorganization of a secondary log. In order to guarantee that every valid log entry is transferred to the new eon, it has to be ensured that each object of a log is processed during an eon. This is done by selecting some logs randomly for reorganization in the first half of an eon and selective in the second half. More details on selection strategies for reorganization can be found in section V.

V. Reorganization

Secondary logs grow over time and require a cleaning policy, also called reorganization, which permanently frees space by removing outdated and deleted (invalid) log entries. The version manager, described in section IV, comes in hand to distinguish invalid from valid log entries required for log cleaning. The complete reorganization process, implemented in DXRAM, is presented in this section.

The reorganization, same as the two-level logging and version control, is designed with the objectives of minimizing RAM consumption without sacrificing overall throughput (s. section VI). In several cases, this is achieved by using the SSD not only for storing the data itself but also for meta-data (section IV) and as a temporary storage (section III). As SSD storage is cheaper than RAM, we think this is reasonable.

To provide more time for the reorganization and also to increase its efficiency, every secondary log is by default twice as large as its backup zone. Furthermore, to allow logging and recovery to be executed concurrently and to reduce the maximal amount of used memory during reorganization, every log is divided into segments of a size of 8 MB (configurable). Therefore, during reorganization of a segment, every other segment can be updated and vice versa. Overall, three threads are involved in logging and reorganization. One network thread at a time writing the received objects to the primary buffer, one writer thread flushing objects from primary buffer to SSD and one reorganization thread. We use a mutex-free implementation with biased prioritization for optimal throughput.

The reorganization process of DXRAM covers four periodically executed steps:

A. Secondary log selection
B. Loading associated version log
C. Segment selection and loading
D. Segment cleaning and flushing back

Steps C and D are repeated several times, e.g. 20 times, to alleviate the overhead of step B.

A. Log Selection

A secondary log is chosen based on its utilization (occupied space to all space ratio). The utilization is a useful metric because all backup zones contain the same amount of payload (except the last one which might not be filled completely). Therefore, the log with the highest utilization has the most invalid data which are outdated or deleted objects. These objects can be discarded to free memory for other backups. The discarding process is called reorganization or cleanup. For the epoch overflow, discussed in section IV, we also enforce each third log selection randomly. Towards the end of an eon all unselected logs, if there are any, are chosen to guarantee the processing of all logs during one eon.

B. Gathering All Versions

A backup peer stores one secondary and one associated version log for each backup zone. The latter contains the current versions of all objects of its corresponding secondary log (data only in secondary log). If a reorganization is to be started for a selected secondary log, its associated version

log is read into a hash table (similar to the version buffer). As we reorganize only one secondary log at a time, the memory consumption is limited, e.g. around 40 MB for 64-byte objects and backup zones with 256 MB payload. After the reorganization has finished, we flush the compacted version log back to SSD. This cleaning ensures that version logs do not contiguously grow and recovery can be performed faster. After flushing, we increment the epoch number.

Access to the version buffer is blocked only for a short period of time, while the epoch number is incremented. During reading all entries into memory and writing back the compacted logs, the version buffer can be filled and read in parallel. In the reorganization process every log entry must be compared to the current version stored in the hash table. In addition, the version buffer is used to check if a log entry has been created in the current epoch.

C. Segment Selection

A secondary log is never reorganized as a whole but incrementally by reorganizing single segments (default: 8 MB). Similar to the secondary log selection, the segment selection tries to find the segment with the most outdated data. The segment selection is on a best-effort basis because determining the segment with the best cost-benefit ratio [14] like in RAMCloud would require to store a timestamp for each object stored in a segment to calculate the segments average age. Furthermore, during updating or removing of an object the previous version's location would have to be known to invalidate the log entry and to update the segment's cost-benefit ratio. This additional meta-data would have to be stored in RAM which is again too expensive for many small data objects. Instead, we calculate a segment's age based on its creation and last reorganization and select the oldest segment for cleaning. We think this is a good metric as there is a higher probability of finding outdated objects in segments that have not been reorganized for a longer period of time. In addition, we also choose segments randomly from time to time and all unselected towards the end of an eon to handle epoch overflows.

D. Segment Cleaning

This phase removes outdated data from a segment using two buffers called old and new buffer (each 8 MB). A selected segment is fully read into the old buffer, all entries are analyzed and valid entries are copied to the new buffer. To check the validity of a log entry, the previously loaded version log and the version buffer (for log entries created within the current epoch) are used. At the end of the segment cleaning we flush the new buffer (containing valid log entries) to SSD whereas the old buffer (containing outdated versions) is freed. We always clean several segments, e.g. 20, of a selected secondary log to not just read the version log for a single segment reorganization.

Deletion of an object is implemented by assigning version -1 in the corresponding version buffer for the CID to be removed. Thus, we do **not** need to write a placeholder (e.g. a log entry without payload and an invalid version number) into the secondary log like tombstones in RAMCloud but only update the version number in the corresponding version log. The marker for an invalid version within the version log is only relevant for the current log entry. All older log entries are implicitly invalidated by the diverging unique version number. Thus, reusing CIDs after removal is safe as any formerly most current log entry will have a different unique version and is therefore always invalid, if it has not already been removed from the secondary log.

VI. Evaluation

In this section we are evaluating the performance of the proposed logging architecture using two benchmarks: YCSB to compare the proposed concepts with Aerospike and Redis and the Log-Cleaner benchmark for comparing with RAMCloud. All benchmark runs were executed on a cluster consisting of 16 identical nodes connected with Gigabit Ethernet. All servers have 16 GB RAM, an Intel Xeon E3-1220 CPU and an Intel SSDSC2CW24 SSD connected via SATA-3 port (350 MB/s write, 500 MB/s read throughput). Debian Jessie with kernel 4.3.0-0 was used as operating system and Java 8 (Oracle) as runtime environment.

Data integrity of DXRAM's logging module was verified several times during the evaluation by recovering all data from SSD and ensuring the number of recovered objects and the object sizes are valid. Furthermore, after every test the log utilizations (objects in log and number of updates during the test) were verified.

A. RAMCloud's Log-Cleaner Benchmark

1) Description: The Log-Cleaner Benchmark was developed at Stanford University to evaluate RAMCloud's two-level log cleaning by measuring the write throughput of a single master under heavy write load [9]. The benchmark utilizes five nodes: One master (stores all data in RAM), three backups (get data from master and store it on disk) and a benchmark client (writes remotely on master). The execution is divided into two phases (loading and benchmarking). In the first phase, the client creates objects on the master with concurrent multiwrite requests until the master reaches a given memory utilization. In the second phase, the client updates objects on the master with a specific distribution (uniform or zipfian) until the cleaning overhead converges to a stable value. Obviously, this workload is unlikely but allows to examine the log reorganization in a worst case scenario. We adopted the C++-Implementation to DXRAM and discovered the maximum number of 100-byte objects that RAMCloud can store on a single server with a log utilization of 80 % and used the same amount of objects for DXRAM for a fair comparison (DXRAM would be capable of storing even more objects). The maximum for our hardware is nearly 52 million objects. To get similar runtimes for the experiments, for the second phase, we used 6 million updates for RAMCloud and 60 million updates for DXRAM. For all comparisons between DXRAM and RAMCloud, 512 objects were aggregated per network message to maximize the update rate and the objects were accessed randomly. Furthermore, RAMCloud was configured (as recommended) to use pipelined RPCs allowing 10 outstanding RPC requests on the client side for improved throughput.

2) Results: As expected, the RAM usage of DXRAM is much more efficient for the small data objects than RAMCloud's (Figure 5). In case of memory management, DXRAM

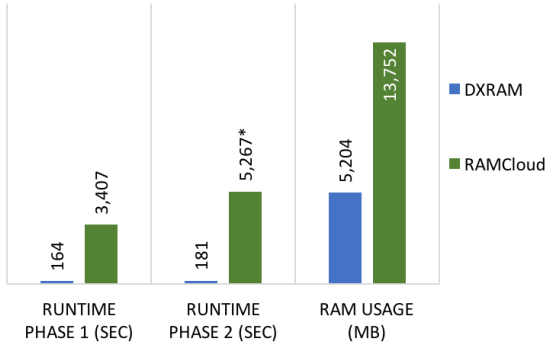

Figure 5: *Log Cleaner Benchmark - RAMCloud vs. DXRAM. *: Extrapolated.*

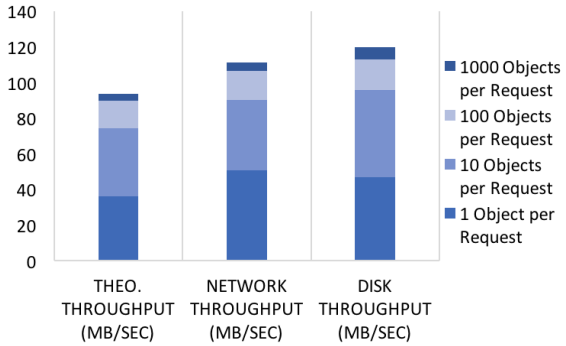

Figure 6: *Log Cleaner Benchmark - Loading Phase with 1 backup peer and 1 master.*

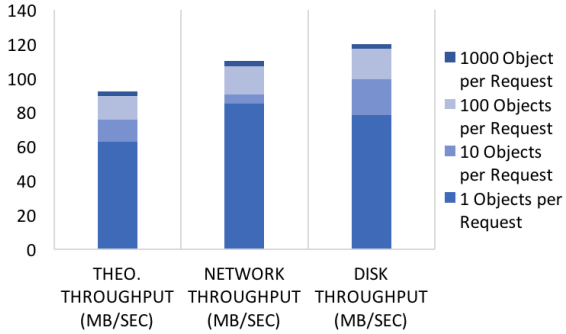

Figure 7: *Log Cleaner Benchmark - Loading Phase with 3 backup peers and 3 masters.*

has an overhead of around 5 % in this scenario, whereas RAMCloud's in-memory log needs more than 250 % memory because RAMCloud is not optimized for small objects. One known bottleneck of RAMCloud is its hash table used for local object lookups which needs more memory than the CID tables of DXRAM (s. section II-B). Still, we think RAMCloud is a good candidate for a comparison as it shares many objectives with DXRAM.

The following experiments were indented to determine, first, the maximum throughput for object creation including backup (phase 1), and second, the maximum update rate (phase 2). Due to the generally limited performance of RAMCloud for small objects (Figure 5), these tests were performed with DXRAM, only. Figure 6 and 7 show the throughput with one master and one backup peer, and three masters and three backup peers respectively. The theoretical throughput presents

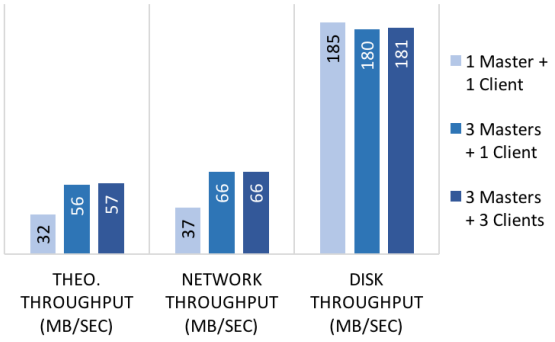

Figure 8: *Log Cleaner Benchmark - Benchmark Phase with 3 backup peers and differing number of masters and clients.*

the runtime divided by the payload size of all objects (object creation rate). The network throughput and the SSD throughput were measured externally on one backup peer during the experiments. Additionally, the number of objects per request were varied. In both scenarios, the network throughput is near the theoretical maximum of Gigabit Ethernet. The small difference is due to the overhead of the memory management and network handler on the masters as all requests were issued by one thread. Therefore, the logging does not slow down the replication process and is as fast as in-memory replication. With more masters and backup peers, the performance is stable for 1,000 objects per request but increased for less objects per request because every created object is replicated three times reducing the management overhead per object.

Figure 8 shows the results for phase 2 where up to three clients update objects on the masters with zipfian distribution. In all three cases, three peers were used for backup. With one master and one client, the master is the bottleneck. For every update request from client the master must send every object to three peers, being limited by network bandwidth. Using three masters, the overall throughput is bound by the backup peers, instead. This is because the SSD bandwidth is shared with the reorganization process when objects are frequently updated. The Log-Cleaner benchmark in phase 2 continuously updates without reading, creating a worst-case scenario for logging. In this scenario the netto write throughput to SSD is around 70 MB/s (including log entry headers and updating version logs) which is good for our HW-setup, as well as compared to RAMCloud. The reorganization tries to free as much memory as possible to prevent the logs from filling up resulting in a higher write throughput. In this experiment the logs were never completely filled but converged to around 90 % utilization. In detail, in phase 2 the SSD is fully utilized with a write throughput of over 180 MB/s as the reorganization must also free 70 MB/s to stabilize the logs' utilization. With a utilization of 90 % around 20 segments (160 MB) and one version log (30 MB) have to be read per second resulting in a read throughput of at least 190 MB/s. Due to the HW limitations parallel reading and writing is very limited. Thus, with a write throughput of 180 MB/s and a read throughput of 190 MB/s we reached the maximum shared raw throughput for mixed read and write operations of the SSD. Consequently, there is not enough bandwidth left to write all incoming backups to SSD without restraining the network threads. To increase the throughput for this worst-case scenario, one can add backup peers to aggregate more disk

bandwidth. In practice, a scenario with all masters updating objects all the time is very unlikely and the logging throughput of many backup peers is aggregated.

As described in section I, RAMCloud tries to bypass the SSD bandwidth limitation by using a log as in-memory data structure and doing reorganization only in memory on the masters. The results of the reorganization are remotely replicated on SSD. In [9] the Log-Cleaner benchmark was executed in a similar setup but with higher network (24 Gb/s Infiniband) and disk bandwidth (700 MB/s). The overall throughput does not exceed 55 MB/s in this experiment for 100-byte objects. In our test, as shown in Figure 8, DXRAM surpasses RAMCloud even with Gigabit Ethernet and less disk bandwidth (70 MB/s). This is surprising and we plan to compare both systems over Infiniband in the future. Overall, DXRAM is inspired by RAMCloud but uses different approaches in many cases for supporting small data objects.

B. Yahoo! Cloud Serving Benchmark

1) Description: The Yahoo! Cloud Serving Benchmark (YCSB) was designed to quantitatively compare distributed serving storage systems [18]. The benchmark offers a set of simple operations (reads, writes, range scans) and a tabular key-value data model to evaluate online storage systems regarding their elasticity, availability and replication. Furthermore, YCSB is easily extensible for new storage systems and new workloads. For our evaluation we used two default workloads A and B and one individual G:

1) Workload A: Ten 100-byte objects per key, 10,000,000 keys, zipfian distribution, 50 % read and write operations, 10,000,000 operations.
2) Workload B: Identical to A, but 95 % read and 5 % write operations.
3) Workload G: One 64-byte object per key, 100,000,000 keys, zipfian distribution, 90 % read and 10 % write operations, 10,000,000 operations.

Aerospike, Redis and DXRAM were configured for same behavior regarding logging and reorganization. In all system all nodes were used as masters and backup peers. For Redis, the total number of objects were decreased as Redis needs more memory than the other systems (Workload A and B: 6,000,000 keys; Workload G: 30,000,000 keys), like described in section I. The impact on the runtime and operation throughput is negligible, the memory usage is extrapolated. In every experiment one version of every object was held in RAM and three versions were replicated and written on SSD. Furthermore, Redis was configured to use append-only logs for persistence, as recommended written once per second. Moreover, the reorganization of aforementioned logs was enabled to prevent the log from constant growing because of invalid (updated or deleted) objects. The re-writing process is triggered when the log size doubled since last re-writing. Aerospike was used with replication to a log file as well. We left the parameters for the compactification of the log file untouched, the sizes for in-memory storage and log file were set properly.

We used 8 storage servers and up to 8 YCSB clients for benchmarking. Each YCSB client was configured to emulate 180 clients using one thread per client (which has shown

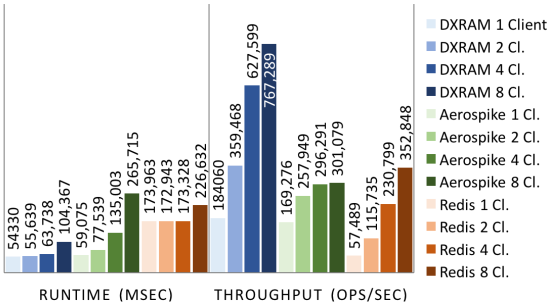

Figure 9: *YCSB - 8 Server, Workload A.*

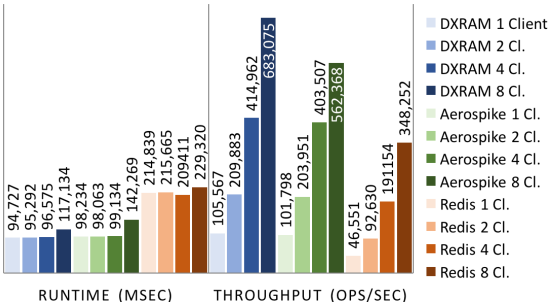

Figure 10: *YCSB - 8 Server, Workload B.*

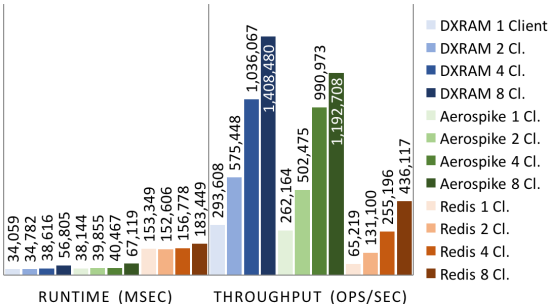

Figure 11: *YCSB - 8 Server, Workload G.*

maximum throughput). Overall, 8 YCSB nodes emulate 1,440 clients.

2) Results: Workload A is a write intensive workload and therefore the best indicator for logging performance. Figure 9 shows that DXRAM outperforms Aerospike and Redis. The numbers also show that the logging approach of DXRAM scales better as for 8 YCSB clients DXRAM is more than twice as fast as the other two systems. Workload B is a read-heavy workload more typical for many graph applications. Figure 10 shows that DXRAM and Aerospike utilize the network perfectly and therefore being close up with up to 4 clients. With 8 clients, Aerospike seems to be once again slowed down by the backup mechanism. Redis is in all cases slower. Workload G is a typical workload for a social media network. The objects are small (64 bytes) and the accesses are dominated by read accesses. The results are similar to workload B for DXRAM and Aerospike. Hence, the small object size does not restrain the systems. Redis on the other hand falls further behind. With 1.4 million operations per second DXRAM is around 15 % faster than Aerospike and more than 300 % than Redis.

Another important aspect revealed by the experiments is that DXRAM needs far less memory compared to the other systems, see Figure 12. Especially for small objects, the

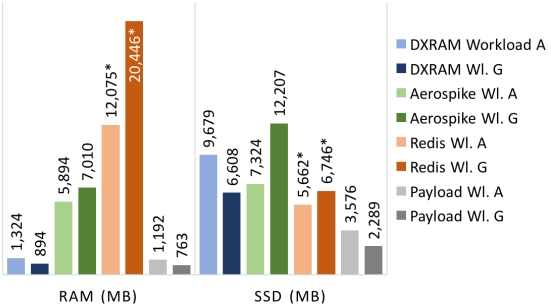

Figure 12: *YCSB - Memory Usage. *: Extrapolated.*

difference is eminently as Aerospike needs more than 700 % the memory and Redis more than 2000 %. The RAM usage values contain the memory management, only, as all system usage is difficult to determine and compare. For optimal performance, DXRAM's logging module adds up to 3 MB per secondary log for version buffer and secondary log buffer. Additional space is required once per backup peer for the primary write buffer (between 16 and 256 MB depending on the workload) and reorganization (maximum around 50 MB for two segment buffers and all versions of one secondary log). Redis on the other hand stores all replicas in RAM and eventually writes them to SSD [7]. Thus, the memory footprint is much higher. In Aerospike the data is written in large blocks on SSD [8] resulting in buffering large amounts of data in RAM. The SSD usage is determined by accumulating the sizes of all files used for backup. In DXRAM every secondary log and every version buffer is one separate file, in Aerospike all data is written to one file and Redis creates one file per slave. The differences regarding the SSD usage between DXRAM, Aerospike and Redis are negligible. For many small objects, DXRAM uses less memory on SSD than Aerospike and the same amount as Redis. For smaller amounts of larger objects, Aerospike and Redis use less memory on SSD. But SSD space is not as valuable as space in RAM because SSD storage is less expensive and easier extendable.

VII. Conclusions

In this paper we proposed a novel log-based replication scheme for many small data objects which provides a high throughput while being memory efficient. The SSD-aware two-level logging approach allows fast persistence and fast recovery for varying application access patterns. Version control is delegated to backup peers avoiding additional memory overhead on master peers. By introducing a version buffer and version log we can provide a fast and memory-efficient version management which, at the same time, makes marker objects for deleted log entries obsolete. Marker objects are known to be difficult to handle in traditional logging approaches.

Experiments with RAMCloud's Log-Cleaner Benchmark demonstrate DXRAM's high object creation and replication throughput and its memory efficiency. The measurements also show a high reorganization throughput of our cleaning policy for small objects (not requiring timestamps for log entries).

The comparison of DXRAM with Aerospike and Redis (both commercial products) using the YCSB benchmark show that the logging and cleaning concepts proposed in this paper allow a higher throughput and better scalability regarding the number of nodes for many small data objects.

Future work includes evaluation and optimization of DXRAM on top of Infiniband including logging, parallel recovery and replica placement strategies. Furthermore, we have already implemented a basic graph processing framework on top of DXRAM which will allow us to study more applications, also being used for evaluating the logging concepts proposed in this paper.

References

[1] R. Nishtala *et al.*, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, Illinois, 2013.

[2] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[3] A. Gulli and A. Signorini, "The indexable web is more than 11.5 billion pages," in *Special interest tracks and posters of the 14th international conference on World Wide Web*. ACM, 2005, pp. 902–903.

[4] B. Fitzpatrick, "Distributed caching with memcached," *Linux journal*, vol. 2004, no. 124, p. 5, 2004.

[5] "Gemfire," http://www.vmware.com/products/vfabric-gemfire/overview.

[6] J. Ousterhout *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[7] S. Sanfilippo and P. Noordhuis, "Redis," 2009.

[8] B. B. C. V. Srinivasan, "A real-time nosql db which preserves acid."

[9] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

[10] F. Klein, K. Beineke, and M. Schöttner, "Memory management for billions of small objects in a distributed in-memory storage," in *IEEE Cluster 2014*, Sep 2014.

[11] D. Ongaro *et al.*, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, New York, NY, USA, 2011.

[12] J. Ousterhout *et al.*, "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010.

[13] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[14] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/146941.146943

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: ACM, 2001, pp. 149–160.

[16] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz, *Handling churn in a DHT*. Computer Science Division, University of California, 2003.

[17] H. Lu, Y. Y. Ng, and Z. Tian, "T-tree or b-tree: main memory database index structure revisited," in *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, 2000, pp. 65–73.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

Chapter 5.

Fast Parallel Recovery of Many Small In-memory Objects

This chapter summarizes the contributions and includes a copy of our paper [14]. A short version of the contributions presented in [14] were transferred to a poster with a peer-reviewed two-page abstract [17].

Paper: Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Fast Parallel Recovery of Many Small In-memory Objects". In: IEEE 23rd International Conference on Parallel and Distributed Systems. Dec. 2017, pp. 248-257

Poster: Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Parallelized Recovery of Hundreds of Millions Small Data Objects". In 2017 IEEE International Conference on Cluster Computing (CLUSTER). Sept. 2017, pp. 621-622

5.1. Paper Summary

While [15] covers the logging mechanism of DXRAM, this publication is focused on the fast parallel recovery of a failed server by replaying the server's remotely logged data. The proposed approach is optimized for hundreds of millions of small data objects but can also be used for large objects. The recovery process is executed on many backup servers in parallel, each recovering a part of the failed server's data, to aggregate disk and network bandwidth. Additionally, on each backup server, the recovery is parallelized to many threads as well to further improve recovery times. Another contribution is the distribution of a server's data into backup zones with a fixed size and replica ordering which is advantageous for the logging as well as the recovery coordination.

The evaluation in the Microsoft Azure cloud with up to 72 instances shows recovery times of under two seconds for servers storing 500 million 64-byte objects, even under heavy load. Furthermore, DXRAM is able to recover failed servers up to nine times faster than RAMCloud.

5.2. Importance and Impact on Thesis

In this publication, we describe the recovery approach of DXRAM which is the second essential component towards a fault-tolerant in-memory key-value store. The evaluation shows that DXRAM is able to mask server failures within seconds under high load. Furthermore, the optimizations regarding the handling of very small objects are visible in comparison with RAMCloud: DXRAM recovers small objects (64 bytes) more than nine times faster than RAMCloud.

The recovery is built upon the remote logging, detailed in Chapter 4, and utilizes an older version of EthDXNet, described in Chapter 3. The backup zone distribution is further discussed in Chapter 6.

5.3. Personal Contribution

All described concepts were developed and implemented by Kevin Beineke, the author of this thesis. Optimizations to the memory management, introduced by Stefan Nothaas, supported the loading of the in-memory storage during the recovery.

The evaluation in the Microsoft Azure cloud was set up and executed by Kevin Beineke.

Prof. Dr. Michael Schöttner and Stefan Nothaas aided the author by providing helpful input in discussions.

Kevin Beineke wrote most of the paper and created all figures. Parts of the introduction, related work and section III.A. and III.B are based on [15]. Prof. Dr. Michael Schöttner and Stefan Nothaas reviewed the paper several times and helped to improve the paper.

Fast Parallel Recovery of Many Small In-memory Objects

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—Social media networks as well as online graph analytics operate on large-scale graphs with millions of vertices, even billions in some cases. Low-latency access is essential, but caching suffers from the mostly irregular access patterns of the aforementioned application domains. Hence, distributed in-memory systems are proposed keeping all data always in memory. These in-memory systems are typically not optimized for the sheer amounts of small data objects, which demands new concepts regarding the local and global data management as well as for the fault-tolerance mechanisms to mask server failures and power outages. In this paper, we propose a novel backup distribution and parallel recovery approach aiming at fast recovery of servers storing hundreds of millions of small objects.**

All proposed concepts have been implemented within the open source distributed system DXRAM and have been evaluated in the Microsoft Azure cloud with up to 72 high performance virtual machines in two scale-sets. For evaluation, we used two benchmarks: the Yahoo! Cloud Serving Benchmark and a recovery benchmark. The experiments show that the proposed recovery strategy is able to recover servers with 500,000,000 small data objects in less than 2 seconds and, also, to efficiently mask server failures under heavy load. Furthermore, DXRAM outperforms the state-of-the-art system RAMCloud in additional recovery experiments with large objects (2.4x faster) and even more with small objects (> 9x).

***Keywords*—Cloud computing; Data centers; Reliability; Storage recovery strategies; Remote replication; Main memory; Flash memory; B-trees;**

I. Introduction

Online graph processing or large-scale social media applications and networks demand low-latency access to billions of very small data objects. On Facebook, for instance, data objects (relationships, likes, shares, status updates etc.) are for the greater part smaller than 64 bytes [1]. Other graph examples are brain simulations with billions of neurons and thousands of connections each [2] or search engines for billions of indexed web pages [3].

Access latency can be reduced by replacing hard disks with flash storage combined with caches in RAM. Due to low-latency data access required by many application domains numerous sophisticated distributed cache solutions have been proposed, e.g. Memcached [1] or Gemfire [4]. While these caches (running in clusters or data centers) successfully reduce access latency, they still burden the programmer to keep caches and back-end storage synchronized which is non-trivial and error prone. Another observation is that caches need to be very large for irregular access patterns. Facebook, for instance, used up to 1,000 Memcached servers to store around 75% of all data in RAM [5]. Still, this approach requires to synchronize updates to Memcached servers with back-end storage systems. To avoid penalties because of cache misses and back-end synchronization, distributed in-memory storages like RAMCloud [6] or DXRAM [7] keep all data always in RAM which is optimal for irregular access patterns seen in many graph processing applications. This design decision requires an efficient memory management to keep meta-data overhead at a minimum as RAM is comparatively expensive.

As RAM is volatile and server failures are the rule not an exception in clouds and large clusters, there is a strong need for fault-tolerance mechanisms. In-memory data needs to be replicated to secondary storage to be able to mask power outages. Hence, in-memory systems provide automatic persistence in the background, in contrast to cache solutions. However, recovering 32 or 64 GB of small data objects of a crashed server might take several minutes which is not acceptable for interactive application domains. The solution is to distribute the data of one server to many backup servers in order to allow parallelization of the recovery process, [8], [9] and [10]. While RAMCloud has shown very fast recovery times of under 2 sec with 60 backup servers for a server storing larger objects, we are focusing in this paper on the recovery of servers storing very small data objects (32 - 128 byte) which is even more challenging.

The contributions of this paper are:

- an efficient range-based backup replica management
- a fast parallel recovery of servers storing hundreds of millions of small data objects
- evaluation of the proposed recovery concepts in the Microsoft Azure cloud showing that a server storing 5×10^8 data objects can be recovered within 2 sec

The evaluation with two benchmarks and a comparison with RAMCloud show that the proposed recovery concepts are very fast, even under heavy load and outperform RAMCloud.

The further structure of this paper is as follows. Related work is discussed in section II, followed by a brief introduction of the key features of DXRAM in section III. Section IV describes the distribution of a server's data. Followed by a section that discusses the recovery coordination. The processes of the local recovery are presented in section VI. The evaluation is discussed in section VII. Conclusions and an outlook on future work are found in the last section VIII.

II. Related Work

Numerous distributed in-memory systems have been proposed to provide low-latency data access for online queries and analytics for various graph applications. Because of space constraints, we can only discuss the most relevant work related to crash recovery of storage servers.

RAMCloud shares several objectives with DXRAM but varies significantly regarding the architecture. RAMCloud provides a table-based in-memory storage to keep all data always in memory. It is designed for larger objects and suffers from a comparably large overhead for small data objects [10]. RAMCloud uses a distributed hash table, maintained by a central coordinator, to map 64-bit global IDs to servers which can also be cached by clients. For persistence and fault tolerance, RAMCloud implements a log-based replication of data on remote servers' disks [9]. In contrast to other in-memory systems, RAMCloud organizes in-memory data also as a log which is scattered for replication purposes across many servers' disks in a master slave coupling [6]. Scattering the state of one server's log on many backup servers allows fast parallel recovery.

In case of a server failure RAMCloud's coordinator must gather the locations of all crashed master's replicas by querying all backups in the system. Next, the coordinator groups the tablets of the failed master into partitions which are assigned to recovery masters. A recovery master coordinates the recovery of its assigned partition by collecting the data from backup servers which read the data from the log stored on SSD or HDD [9]. Furthermore, the recovery masters must reconstruct a hash table for the failed master's key space in order to make the objects available again. In contrast, DXRAM's superpeers know the locations of all backups a-priori and initialize the recovery instantaneously. Thereupon, the backup servers recover the data from SSD and update the meta-data on the superpeers by sending a list of the recovered objects aggregated to more compact ID ranges.

Google's **Bigtable** is a distributed storage system which is used for web indexing, Google Earth and many more services [8]. A Bigtable is a distributed sorted map which assigns a row key, an arbitrary string, a column family that defines access control and a time stamp to every value. A Bigtable cluster consists of a number of tables whereas each table comprises a set of tablets. Then again, a tablet contains the data of a row range and is around 100 to 200 MB in size. The tablets are distributed to several servers within the cluster to aggregate storage and also to enable parallel recovery. Each Bigtable server stores all update operations in a single log to bundle writes and therefore improve the access to persistent storage devices. However, this requires another step during recovery in which the log has to be sorted in order to recover the data of one tablet. DXRAM on the other side uses one log per backup zone to avoid this sorting step and ensures efficient accesses to flash storage by its two-level logging approach [11].

Aerospike is a distributed database platform providing consistency, reliability, self-management and high performance clustering [12]. Aerospike uses Paxos consensus for server joining and failing, and balances load with migrating of partitions. DXRAM also offers migration but at fine-grained object level. The object lookup is provided by a distributed hash table in Aerospike. Aerospike is optimized for TCP/IP. Additionally, Aerospike enables different storage modes for every namespace. For instance, all data can be stored on SSD with indexes in RAM or all data can be stored in RAM and optionally on SSD with a configurable replication factor. As Aerospike is a commercial product, not many implementation details are published. Aerospike does not offer a possibility to recover servers during ongoing operation but provides data restore on cold-start.

Redis is another well known distributed in-memory system which can be used as an in-memory database or as a cache [13]. Redis provides a master-slave asynchronous replication and different on-disk persistence modes. To replicate in-memory objects, exact copies of masters, called slaves, are filled with all objects asynchronously in remote memory. This is quite expensive as slaves need to provide as much RAM as the servers they need to backup. To overcome power outages and server failures, snapshotting and append-only logging with periodical rewriting can be used. Like Aerospike, Redis offers a recovery on startup, only. Further, the recovery is not able to recover one server in parallel from different slaves.

Alluxio is a distributed file system which provides fast data access times for all objects in cluster setups [14]. This is achieved by holding all objects in RAM and avoiding replication to other servers and slower secondary storage through a lineage-based approach. In case of a server failure, the data is reconstructed by re-executing the operations that generated the data. The re-computation overhead is limited by additional asynchronous checkpointing to remote disks.

While object creations and updates benefit from the replication-less approach, the recovery is impaired for high throughput scenarios as checkpointing falls behind (bound to I/O bandwidth). As a consequence, many objects have to be reconstructed based on possibly many jobs which have been executed since the last completed checkpoint. This reduces recovery throughput, especially for many small objects. Additionally, storing the job binaries for a long time increases the RAM consumption.

III. DXRAM Architecture Overview

DXRAM is an open source (*https://github.com/hhu-bsinfo/dxram*) distributed in-memory system with a layered architecture, written in Java. It is extensible with additional services and data models beyond the key-value foundation of the DXRAM core [10]. In DXRAM, an in-memory data object is called a *chunk* whereas an object stored in a log on SSD is referred to as *log entry*.

A. Global Meta-Data Management

In DXRAM, every server is either a peer or a superpeer. Peers store chunks, may run computations and exchange data directly with other peers, and also serve client requests when DXRAM is used as a back-end storage. Peers can be storage *servers* (with in-memory chunks), *backup servers* (with logged chunks to SSD) or both. Superpeers store global meta-data like the locations of chunks, implement a monitoring facility, detect failures and coordinate the recovery of failed peers, and also provide a naming service. The superpeers are arranged in a Chord-like overlay [15] adapted to the conditions in a

data center. Moreover, every peer is assigned to one superpeer which is responsible for meta-data management and recovery coordination of its associated peers. During server startup, every server receives a unique node ID employing ZooKeeper.

Every superpeer replicates its data on three succeeding superpeers in the ring. If a superpeer becomes unavailable, the first successor will automatically take place and stabilize the overlay. In case of a power outage, the meta-data can be reconstructed based on the recovered peers' data. Thus, storing the meta-data on SSD on superpeers is not necessary. Superpeer failures will not be further discussed in this paper.

Every chunk in DXRAM has a 64-bit globally unique chunk ID (CID). This ID consists of two separate parts: a 16-bit node ID of the chunk's creator and a 48-bit locally unique sequential number. With the creator's node ID being part of a CID, every chunk's initial location is known a-priori. But, the location of a chunk may change over time in case of load balancing decisions or when a server fails permanently. Superpeers use a modified B-tree [16], called lookup tree, allowing a space efficient and fast server lookup while supporting chunk migrations. Space efficiency is achieved by a per-server sequential ID generation and ID re-usage in case of chunk removals allowing to manage chunk locations using CID ranges with one entry for a set of chunks. In turn, a chunk location lookup will reply with a range of CIDs, not a single location, only. This reduces the number of location lookup requests. For caching of lookup locations on peers, a similar tree is used further reducing network load for lookups.

B. Memory Management

The sequential order of CIDs (as described in section III-A) allows us to use compact paging-like address translation tables on servers with a constant lookup time complexity. Although, this table structure has similarities with well known operating systems' paging tables we apply it in a different manner. On each DXRAM server, we use the lower part (LID) of the CID as a key to lookup the virtual memory address of the stored chunk data. The LID is split into multiple parts (e.g. 4 parts of 12 bit each) representing the distinct levels of the paging hierarchy. This allows us to allocate and free page tables on demand reducing the overall memory consumption of the local meta-data management. Complemented with an additional level indexed by node ID storing of migrated chunks is possible as well. DXRAM uses a tailored memory allocator with very low footprint working on a large pre-reserved memory block [10]. For performance and space efficiency reasons, all memory operations are implemented using the Java Unsafe class.

Chunks store binary data and each chunk ID (CID) contains the *creator*. Chunks can be migrated to other servers for load balancing reasons. Migrated chunks are then called *immigrated chunks* on the receiver and *emigrated chunks* on the creator. Finally, there are *recovered chunks* stored on a new *owner* after a server failure.

C. Remote Logging

Each chunk is replicated asynchronously to multiple remote SSDs for fault tolerance reasons. We use an active replication approach with predefined replica ordering. For the expected

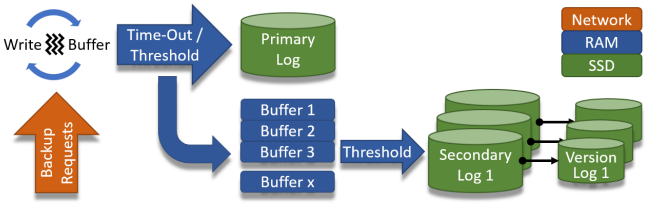

Figure 1: *Logging Architecture*

large amount of chunks, storing all replicas in RAM is too expensive and traditional HDDs cannot satisfy the applications' demands on low latency and throughput. Furthermore, replicas do not serve client requests as reading from SSD is too slow. Writing chunks to SSD with in-place updates requires a lookup data structure either stored in RAM or on SSD which is both not efficient. Holding the locations of all replicas for billions of chunks in RAM results in a far too large memory overhead and accessing SSD to determine the location of an chunk prior to each write access is too slow. Therefore, we decided to store chunk backups in logs on SSD (figure 1) which allows high throughput due to sequential appends.

We use a **two-level logging** approach allowing fast recovery and fast persistence with high throughput: fast recovery is provided by sorting incoming backups per server into different *secondary logs*. This reduces the amount of log entries to be analyzed during recovery. Fast persistence and high throughput are conflicting objectives in case of small backup requests. Forcing small backups, e.g. 64 bytes, directly onto SSD would result in low throughput. Obviously, buffering and aggregating small backups to be written in larger bundles helps to improve throughput but delays persistence. We address these challenges by buffering and using an additional log, called *primary log*. It is likely that a backup server receives many backup requests from different servers which are bundled in a buffer and flushed into the primary log (without sorting). This ensures fast persistence and in the unlikely case of backup idle phases, we flush buffers based on a timeout, too. Backups flushed to the primary log are kept in memory to be flushed later to their associated secondary log, when enough backups from one server have been collected. We assume typical battery backup is available on all servers allowing to always flush data to the primary log in case of a power outage. If backup requests are large, e.g. 4 KB or more, they are directly flushed to their corresponding secondary log. Furthermore, the incoming buffer is sorted by secondary log to speed-up bundling backup requests which have to be written into the same secondary log.

Every instance of a backed-up chunk needs a version number for validation during reorganization and recovery. DXRAM uses a **backup-side version control** which holds only the most recent versions in RAM and stores all other versions in one version log per secondary log on SSD. In order to avoid overhead of globally applicable versions, the most current log entry of all its backup servers is always found through an enforced replication ordering. We use an epoch-based approach for assigning and resolving version numbers providing a low memory footprint and high throughput [11].

Many graph processing applications are read heavy and updates are rather seldom [17]. Still, a secondary log must be reorganized if updates and deletes reach a threshold to avoid

filling it up with outdated and deleted versions. For that purpose, a log is split into segments (default: 8 MB each) enabling incremental processing. For increased efficiency, the log with most invalid data is chosen for reorganization. Subsequently, all version numbers, which are stored en-bloc in a version log, are read from SSD and a predefined number of segments is reorganized. In that process, each chunk is validated against the read-in version information and all valid chunks are written back to SSD, omitting the invalid ones.

D. Recovery Overview

This section summarizes the full recovery process which is presented in more detail in the following sections including optimizations.

As chunks are replicated to SSDs on remote servers, the recovery performance on a single server is limited by its hardware. Thus, like RAMCloud and Google's Bigtable, DXRAM scatters the chunks from one server to many backup servers to aggregate SSD bandwidth and CPU processing power. Backup servers are not determined for each chunk but for **backup zones**, containing up to 256 MB of chunks, to minimize meta-data overhead for backups. Hence, a server's data is split into 256 MB blocks which can be recovered from a backup server within 1 to 2 seconds. This process can be performed by many backup servers in parallel allowing high scalability. For every backup zone, three backup servers are assigned with a fixed replication and recovery ordering. In order to avoid a broadcast, superpeers store the backup zones of each of their associated peers. Thus, they can coordinate recovery and directly contact the correct backups in case of a failure. However, they do not need to store CIDs per backup zone; this information is needed by servers, only. Network limitations are masked by recovering a backup zone in the memory of backup servers and resume normal operation. Chunks can be migrated asynchronously to a fresh server later. Further discussion on the backup zones can be found in section IV-A.

Every write access to a chunk (create, delete and update) is replicated to the backup servers of the particular backup zone, according to the replication ordering. To efficiently resolve the backup zone affiliation for billions of locally stored chunks, which is necessary to send the replicas to backup servers, every DXRAM server utilizes a B-tree which is optimized for storing CID ranges. This **backup zone tree** provides fast access times while being very space efficient because of range aggregation (e.g. an entire backup zone with millions of chunks can be stored with 1 to 2 entries within the B-tree). The architecture and usage of the backup zone tree is described in section IV-B.

Server failures are detected and recovery is coordinated by the superpeer next in the superpeer overlay. This superpeer informs the responsible backup servers storing relevant backup zones. Then, the backup servers recover all valid chunks from the associated secondary logs into their local memory. All required information to initialize the recovery of a failed server is available a-priori on the superpeer as backup servers of all backup zones are stored on superpeers (including backups) as well. Thus, there is no need to gather information from backup servers (in contrary to RAMCloud and Google's Bigtable).

The local recovery on a backup server is also challenging as a typical secondary log stores several millions of small log entries and for every single log entry the validity (currentness and status) and correctness (data integrity) has to be verified. To limit the temporary memory consumption, a secondary log is recovered segment by segment. The segments are processed by iterating over all log entries and restoring the valid log entries. The validity of a chunk is verified by reading all current version numbers from SSD (stored in a version log) before the recovery process and comparing them with the log entry version numbers. Obviously, gathering, storing, reading and comparing millions of version numbers is time critical. Furthermore, parallelization is crucial to speed-up the recovery process and increase the overall system's performance and responsiveness by improving the availability of chunks. Our concepts for a highly efficient recovery of entire backup zones are presented throughout section VI.

Finally, the lookup meta-data of all recovered chunks must be updated on corresponding superpeers. The necessary network transfer can be minimized by aggregating CIDs into ranges, see section VI-C.

IV. Backup Zones

In order to enable a fast parallel recovery, the chunks of one server are partitioned into several backup zones which are scattered across potentially many backup servers (e.g. a 64 GB server assigned with 256 different backup servers). Every server determines its own backup zones, e.g. randomly, and informs its associated superpeer on each backup zone creation. This approach avoids global coordination regarding backup zone selection between servers. We use a replication factor of three by default but it is configurable.

A. Local Backup Zone Management

Each backup zone is identified by a zone ID (ZID). The ZID alone is not globally unique but it is in combination with the creator's node ID derived from the context. A new backup zone is created whenever a chunk does not fit into any existing backup zone. If chunks were deleted, a backup zone will be gradually refilled with new chunks. Furthermore, chunks with reused CIDs are stored in the same backup zone as before, if possible, to minimize meta-data overhead (see IV-B). Three backup servers are assigned to each backup zone with a fixed replication ordering guaranteeing consistency. According to the ordering, the first backup server receives all backup requests first, the second afterwards and so on. Furthermore, backup requests are bundled whenever possible. If there are less than three servers currently available for backup (e.g. during startup), the next joining server will be used and receives all previously replicated chunks of this zone. All backup servers are chosen randomly, optionally with disjunctive first backup peers. Other replication schemes like copyset replication [18] or location aware approaches [19] can be used, too.

A server notifies its superpeer whenever a new backup zone was created or a backup server was changed. This results in a single message for every 256 MB (e.g. once after 3.5×10^6 64-byte chunks have been created) and a few messages per server failure (the failed backup server has to be replaced), only. To further reduce memory consumption on superpeers (resulting in just 10 bytes per backup zone in the best case), a superpeer does not store backup zone affiliations of chunks.

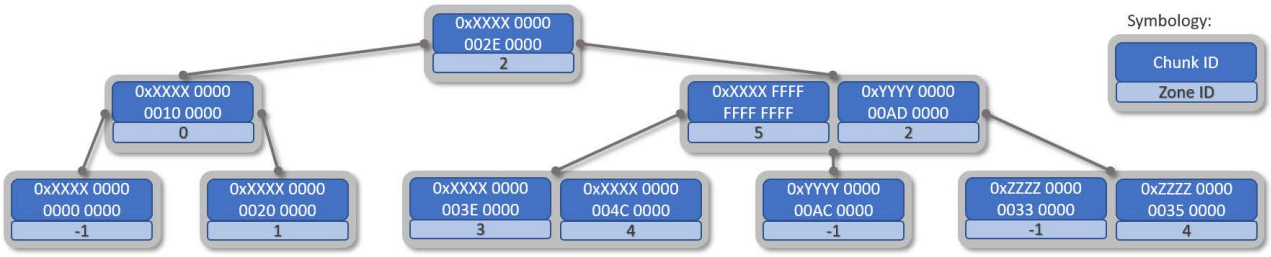

Figure 2: *Backup Zone Tree with order 4 - This tree stores a total of 6 backup zones: the first contains the locally created chunks 0xXXXX 0000 0000 0001 to 0xXXXX 0000 0010 0000 (creator has the node ID 0xXXXX), the second 0xXXXX 0000 0010 0001 to 0xXXXX 0000 0020 0000, the third 0xXXXX 0000 0020 0001 to 0xXXXX 0000 002E 0000 with the immigrated chunks 0xYYYY 0000 00AC 0001 to 0xYYYY 0000 00AD 0000. The fourth contains* 2^{20} *locally created chunks, the fifth contains locally and immigrated chunks, again. The sixth backup zone is not yet concluded. The zone ID* -1 *defines the end of CID ranges whose chunks are not stored on this server.*

This information is exclusively stored on the owner of a chunk as only this server must know the corresponding backup zone of its chunks for sending backup updates.

B. Backup Zone Tree

A backup zone might consist of locally created, immigrated and recovered chunks. To store the backup zone affiliation of every chunk, we use a B-tree similar to the lookup tree, see [20]. This tree is called *backup zone tree* and stores (beginning chunk ID, end chunk ID, zone ID) tuples. A tree range can be equal to one backup zone but only if all its CIDs are consecutive (if all containing chunks were created on the same creator and all deleted chunks' CIDs were re-assigned, if at all, to the particular backup zone).

One tuple is typically stored with two entries in the backup zone tree (see figure 2). One for the beginning of the range and one for the end. Directly succeeding ranges can be stored with one entry per range, only, as the end of one range defines the beginning of the next one. Limited to locally created chunks, there is just one tuple per backup zone in the tree resulting in a total of 3 to 4 KB of meta-data for 64 GB servers storing nearly a billion 64-byte chunks. The memory consumption depends also on the alignment and fill rate of the backup zones, the order of the B-tree (default is 10) and access pattern. Additionally, searching is very fast because the backup zone tree has a height of 2 to 3, only, for a billion local chunks and an order of 10, for instance.

The CIDs of locally created chunks are not registered separately in the backup zone tree, but as one range as soon as the current backup zone is completely filled. The current backup zone's end is thereby represented by the highest possible CID (0xXXXX FFFF FFFF FFFF, where 0xXXXX is the node ID of the server) and is replaced by the currently highest used CID on creation of a new backup zone. Therefore, the end of the current backup zone is always identified by the highest used CID and the end of every preceding backup zone is registered in the tree.

Emigrated and deleted chunks within the registered backup zones can be ignored because they are no longer relevant. This reduces the memory consumption noticeably as expensive range splits are prevented [20]. For instance, removing a single CID from a CID range would result in three ranges: first, all CIDs from beginning up to the removed CID - 1. Second, the removed CID and third, the CIDs up to the end.

On the contrary, it is imperative for replication purposes to store the backup zone for immigrated and recovered chunks. Immigrated chunks are registered range-wise if possible and chunks of a recovered backup zone are added all at once for efficiency reasons (see V-C).

V. Recovery Coordination

A. Failure Detection

Fail-stop server failures are detected based on the superpeer overlay as every superpeer pings each of its peers periodically (configurable interval). If any message (not limited to ping messages) could not be delivered, an error occurred during incoming or outgoing transmission or a response is missing, the failure detection is started. This triggers either a *ResponseDelayedEvent* or a *ConnectionClosedEvent* based on the detected error. A *ResponseDelayedEvent* is less significant than a *ConnectionClosedEvent* as it occurs whenever a response is delayed. The cause can be diverse and is not restricted to a network problem. If a *ResponseDelayedEvent* is triggered, a message will be sent to the previously unresponsive server. If the server receives the message, the event is ignored. Otherwise a *ConnectionClosedEvent* will be triggered automatically as the connection is either closed or the transmission is about to fail. A *ConnectionClosedEvent* on the other side is handled by creating a new connection to the supposedly failed server. If the connection could be successfully established, the prior disconnection is ignored. Otherwise failure handling is started. Depending on the application access pattern, a peer might detect the failure earlier than the corresponding superpeer. In this case, the peer informs the superpeer to speed-up failure detection and thus the whole recovery process.

Performance or omission failures need additional handling: if a server is only temporarily unavailable, it will have to join again. The superpeer can check if the server's data was already recovered and, if necessary, command the corresponding servers to roll back by writing the recovered chunks to SSD. The server re-joining is finished when all backups are stored in the same logs as before the incident. Chunks that have been updated prior to the re-joining are included as they have been written to the memory management of the replacement servers. Updates during the re-joining are postponed.

Failure handling affects many parts of DXRAM, but only backup- and recovery-relevant aspects are discussed here.

Peers and superpeers respond differently to a peer failure. Peers replace the failed peer in every affected backup zone by new backup servers and replicate the chunks accordingly to maintain the replication factor. Superpeers, on the other hand, notify all peers by propagating the server failure to all superpeers first and then to all of its peers. This triggers the failure handling on each affected peer. Then, the superpeer responsible for the failed peer initializes the recovery.

B. Initializing Parallel Recovery

To initialize server recovery, the responsible superpeer sends one *RecoverBackupZoneRequest* to the first backup server of each backup zone of the failed server. The recovery of the backup zone is then executed by the backup server. If the first backup server is unavailable, the second will be notified and so on. The superpeer sends all requests for all backup zones of the failed server at once for maximal parallelization and collects the responses after the recovery is finished. If a response is missing, the recovery is initialized on the next backup replica of that backup zone. The recovery of a backup zone using the logs is described in section VI.

C. Finalizing Parallel Recovery

When all contacted backup servers reported recovery success, the superpeers must update its meta-data to make all recovered chunks available again. Therefore, the lookup trees of the chunk creators (divergent from failed server for immigrated and previously recovered chunks) are modified on the corresponding superpeers by adding CID ranges complemented by the *restorer* of the backup zone which becomes the new owner. For that purpose, the CIDs of the recovered chunks must be aggregated to ranges beforehand. This is discussed in subsection VI-C. At this point all chunks are available again.

The meta-data of the failed server's backup zones are deleted on the corresponding superpeers as they are not needed anymore. To administer recurring failures, all servers that recovered a backup zone of the failed server have to create a new backup zone for the recovered chunks. Therefore, the corresponding superpeer must be notified and the local backup zone tree must be updated as well. Moreover, all backup servers of the new backup zone must receive all recovered chunks for backup. To avoid sending recovered chunks three times to three new backup servers, the backup servers of the old backup zone are reused. Thus, only one new backup server must be determined if no additional failure occurred (one backup server cannot be used as it is the restorer). Obviously, the new backup zone must contain the same set of chunks.

The locality within backup zones is maintained during the recovery but, in some cases, it could be reasonable to fully reconstruct the original server by sending all recovered chunks to a new server, which will increase recovery time but can be done concurrently to minimize availability interruption.

VI. Local Recovery of Many Small Chunks

The local recovery must be highly optimized to enable fast recovery of millions of small chunks. For instance, a 256 MB backup zone may consist of more than 3.5×10^6 chunks with an average payload size of 64 bytes per chunk. This results in the associated secondary log storing between 3.5×10^6 and 7×10^6 chunks depending on the backup rate and efficiency of the reorganization (a secondary log is double the size of the backup zone by default).

A. Overview

At the beginning of the local recovery, all corresponding log buffers must be flushed to guarantee that every recoverable chunk is stored in the associated secondary log. Then, the version log for this secondary log is loaded from SSD for fast access (see VI-B). The recovery is executed segment by segment as follows: first, a segment is read into a byte buffer (default segment size: 8 MB). Next, every chunk is analyzed by iterating over the byte buffer. The analysis includes validation and error detection. To validate a chunk, the read-in version number is compared to the version number stored within the log entry header of a chunk. If the chunk is invalid (outdated or deleted), it will be ignored. The error detection is optional and uses a 32 bit cyclic redundancy check.

Valid and error-free chunks are then stored to the local memory management of DXRAM. Small chunks are bundled in batches up to 100,000 chunks to benefit from fast batch allocation of the memory management. Furthermore, the chunks are not copied, but the byte buffer is passed to the memory management along with an index buffer containing all CIDs, offsets within the segment buffer and lengths of the chunks. At the end, the secondary log is removed from SSD.

B. Version Log

During recovery, every chunk read from secondary log must be validated to avoid recovering outdated and deleted chunks. For efficiency reasons, we store all version numbers for each secondary log in a separate version log (see figure 1). Version logs are also compacted during cleaning of their corresponding secondary log [11]. This approach allows us to read-in only relevant version numbers for log cleaning and recovery ensuring fast processing and low memory footprint. For recovery of a backup zone, the corresponding version log is read from SSD and all version numbers are added to an array or a hash table. The array is optimal for storing version information for larger CID ranges which are heavily used in DXRAM and thus likely to occur. For scattered CIDs, we dynamically switch to the hash table.

Backup zones contain chunks that were created by the backup zone creator but also immigrated for load balancing reasons or recovered from a failed server. Accordingly, the CIDs could be rather arbitrary within one backup zone. But, typically, a backup zone contains one large range (all chunks from one creator) with a few chunks outside of the range (such as migrations for load balancing or chunks with reused CIDs stored in another backup zone). Therefore, it is beneficial to determine large ranges within a backup zone, whose versions can be stored in the array, but only if it is lightweight enough to not burden the logging throughput. We calculate the arithmetical mean of all CIDs incrementally with every write access. Every version number of a chunk with a CID whose euclidean distance is smaller or equal half the average number of chunks per backup zone is stored in the array with its CID as index. All other version numbers are stored in the hash table. For the validation process, the same metric is used

to decide whether to look in the array or in the hash table for the current version number of a chunk. Therefore, the benefit of faster access to the array is two fold.

To avoid that the average CID value drifts off because of emigrated chunks with much higher or lower CID, all CIDs with a large euclidean distance to the average value are ignored. This mechanisms is delayed to reduce the impact of early updates of emigrated chunks. This is a best-effort approach limiting performance to a hash table level in worst case, but improving performance considerably in other, more likely, cases without burdening the CPU.

C. CID Range Determination

After executing the local recovery, the backup servers notify the superpeers by sending a list with the CIDs of all recovered chunks. This information is used to update the lookup trees of the superpeers to make recovered chunks available again. A typical backup zone consists of several million small chunks. Sending millions of 8-byte CIDs would result in large message sizes (for 64-byte chunks close to 26 MB) and slow processing (every CID must be processed locally). But, backup zones typically store large CID ranges and a few migrations and, usually, the migrations are also aggregated. Therefore, we can combine CIDs into ranges during the recovery process.

The CID range determination requires a sorted list of all CIDs. Gathering all CIDs during recovery by adding them with insertion sort, for instance, is too time-consuming. Instead, the already available data structures for the validation are used.

The `array` uses the CID as index by subtracting a non-varying offset. Thus, it is already sorted but might contain gaps for chunks that were deleted (and whose CIDs are not yet reused).

The `hash table` is based on an array as well, but the array stores (chunk ID, version number) tuples instead of version numbers only. Moreover, the tuples are not sorted by CID as the CIDs are hashed before insertion. To sort the array, two different approaches are used. If the hash table is barely filled and thereby contains many empty array entries, insertion sort will be used. If the hash table is well-filled (threshold: 100,000 entries), quicksort will be used.

The ranges can be determined by iterating the sorted array and adding the begin and end of every consecutive sequence to another list. Subsequently, the ranges are sent to the superpeers and stored in their lookup trees.

D. Parallelization of the Local Recovery

While the recovery is executed, all chunks of the failed server are unavailable. Therefore, the recovery has highest priority and is allowed to use all system's resources. Unfortunately, some recovery steps cannot be parallelized because of I/O limitations (e.g. loading the version numbers and segments from SSD) or enforced sequential processing (e.g. storing the chunks in memory management). Therefore, parallelization is focused on analyzing log entries which includes CPU and memory intensive steps like reading the log entry headers, getting and comparing the version numbers, calculating and comparing the checksums, assembling large chunks (see section VI-E) and forwarding gathered data to memory management.

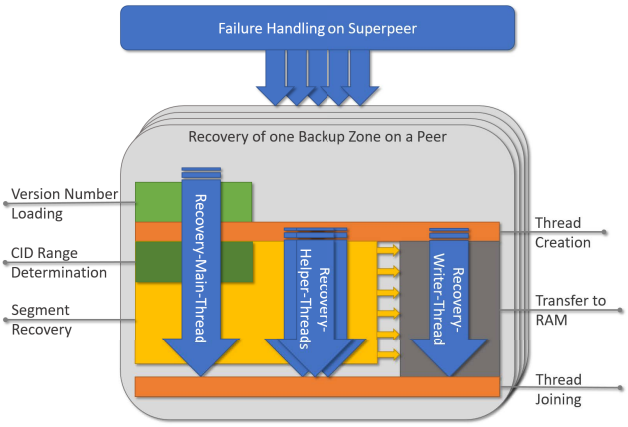

Figure 3: *Parallelization - Thread flow chart for the recovery.*

Additionally, a dedicated thread writes all recovered chunks into memory management. DXRAM uses three thread types for the recovery (fig. 3):

The *RecoveryMainThread* reads all version numbers (hardly parallelizable), creates all other threads, determines the CID ranges, recovers segments (includes analysis of log entries) and returns once the other threads have finished.

At least one *RecoveryHelperThread* supports the *RecoveryMainThread* with recovering segments. The number of *RecoveryHelperThreads* is configurable to adapt to the hardware. Our tests showed a peak recovery performance with 4 *RecoveryHelperThread*.

The *RecoveryWriterThread* receives the read and verified chunks from the other threads and writes them into the memory management by utilizing batch processing mechanisms.

E. Large Chunks

Although we expect primarily small chunks for our target application domains, large chunks (e.g. larger than 1 MB) can be handled, too. Without modifications, chunks larger than one segment cannot be processed. Furthermore, fragmentation can reduce the maximal chunk size even more: after reorganization of a fully utilized backup zone the corresponding secondary log is half-full (as a secondary log is double the size by default to increase efficiency of the reorganization). Without fragmentation, half of the segments are filled completely, the other half is empty. With largest possible fragmentation, every segment is half-full. Accordingly, chunks larger than half a segment cannot be stored without splitting as larger chunks might not fit with adverse fragmentation, even after reorganization. The segment size is configurable, but increasing it to match large chunks is not an option as, first, the maximal chunks size is not always known a-priori and, second, large segments raise the temporal memory consumption during reorganization and recovery.

To allow storing larger chunks, chunks are split into parts of the size segment length / 2 (e.g. 4 MB for default configuration). Every part of a large chunk is one link in a chain and possesses a complete log entry header complemented by a chain ID (position within the chain) and the number of all chain links. Thus, every chain link can be identified, validated and error-checked without the other chain links.

When analyzing a chain link during recovery, a byte buffer large enough (more later) to store all chain links is created and the first link is copied at the correct position within the buffer (chain ID $\times$ segment length / 2). Moreover, the byte buffer and the CID is registered in a hash map. For every additional link, the byte buffer is fetched from the hash map and the link is copied analogously. The byte buffer can be sized accurately if the first occurring link is the link with highest chain ID because the last chain link is the only link with a possibly different size. Chunk size: (chain length - 1) $\times$ segment length / 2 + size of last chain link. Otherwise the size is estimated to the upper bound: chain length $\times$ segment length / 2. The byte buffer's limit is then set, once the last chain link is read (avoiding copying of the buffer). At the end of the recovery, after recovering all segments, all large chunks are written sequentially into memory management.

VII. Evaluation

In this section, we present the evaluation of the proposed recovery architecture using a recovery benchmark and YCSB, and also compare it with the recovery of RAMCloud. All benchmark runs were executed in Microsoft's Azure cloud in Germany Central with up to 72 virtual machines from the type *Standard_DS13_v2* which are memory optimized servers with 8 cores (Intel Xeon E5-2673), 56 GB RAM, 112 GB SSD capacity (maximal cached throughput: 256 MBps) and 5 GBit/s Ethernet connectivity. In order to manage the servers, we created 2 identical scale-sets (as one scale-set is limited to 40 VMs) based on a custom Ubuntu 14.04 image with 4.4.0-59 kernel and a third scale-set with Debian 8 image and 3.16.0-4 kernel for RAMCloud (hardware and configuration identical). We want to give thanks to Microsoft for providing us with a Azure research sponsorship.

A. Recovery Benchmark

1) Description: The recovery benchmark is used to study the parallel recovery of one server. In this test, the server creates 5×10^8 64-byte chunks with a total payload of more than 30 GB. Because of resource constraints of our sponsorship, we could not use more than 72 backup servers. In order to study the maximum recovery throughput we limit payload to 30 GB (resulting in 144 backup zones, each 256 MB) although the server could handle more. As a result each backup server stores at least two backup zones as the data is replicated to x slaves, with $x \in [1, 72]$ during this test. After logging all chunks, the master is killed which initiates the recovery process for all 144 backup zones. The used backup zone distribution strategy chooses disjunctive first backup peers to enable maximum parallelism during recovery. If all backup peers are used as first backup peer already, the procedure is repeated.

2) Results: The logging performance is at maximum by utilizing the entire I/O capacity (figure 4). With up to 4 backup peers, the SSDs of the backup peers are the bottleneck. With more aggregated SSD bandwidth, the network becomes the limiting factor (> 500 MB/s).

As expected, the recovery times improve with the number of backup peers (see figure 4). With 72 backup peers (2 backup zones per backup peer) the complete recovery process takes less than 2 seconds resulting in a recovery throughput of more than 16.5 GB/s. The recovery process includes failure handling, ZooKeeper cleanup (to enable node ID re-usage), peer failure propagation, superpeer overlay cleanup, recovery initialization and finalization, and meta-data updating on the superpeer. Additionally, all peers replace the failed peer in every backup zone and recover all chunks from SSD. Re-replication of the failed server's chunks is excluded and executed concurrently afterwards, which does not impair availability as every object is accessible after recovery, already.

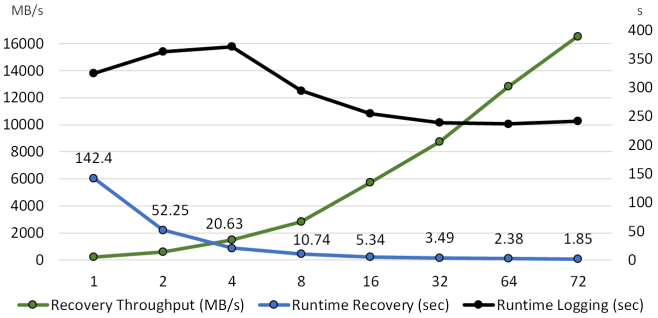

Figure 4: *Overall Recovery Performance*

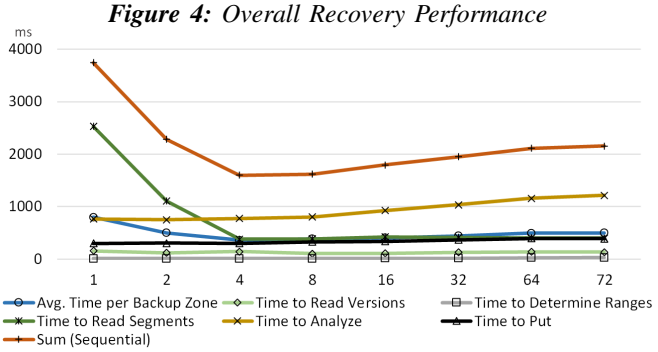

Figure 5: *Recovery Performance in Detail*

Figure 5 shows the average local recovery times and also the sequential times of all of its steps. The blue line shows the average local recovery time achieved by parallelizing all other steps resulting in about 500 ms for the local recovery of one backup zone. The sum of all single steps is significantly higher than the average recovery time as many threads work in parallel during recovery increasing recovery performance substantially. The local recovery of a single backup zone (blue line) is reduced by up to 400 ms when using four or more backup servers which can be explained by kernel buffer caching effects on backup servers: the backup servers may still have log entries in their caches when the recovery is started and hence read some log entries from cache not from SSD. With more than 4 backup peers the times to analyze all log entries (yellow line) and thus the single backup zone recovery times grow slightly as runtime optimizations are less effective or not applicable with fewer iterations per backup peer.

The evaluation with the recovery benchmark shows that DXRAM is able to recover 5×10^8 64 byte data objects in parallel in less than 2 seconds. We expect even better recovery times if we could use more backup peers.

B. A Comparison with RAMCloud

1) Description: Only a few distributed in-memory systems provide a parallel live recovery whereof RAMCloud is a

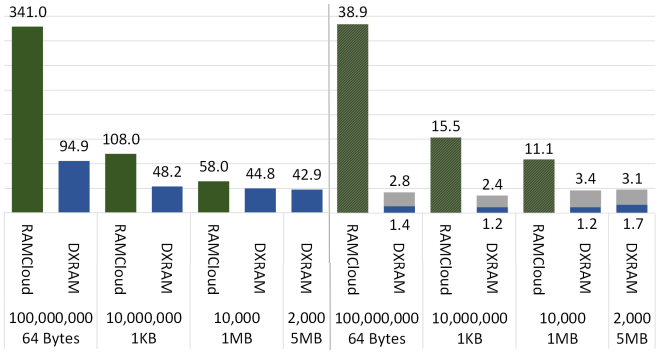

Figure 6: *Logging (left) And Recovery (right) Performance of RAMCloud and DXRAM. Runtime in sec.*

prominent example providing an own recovery benchmark. We configured both recovery benchmarks in the same way: one master server generates up to 10^8 objects with sizes between 64 bytes and 5 MB and replicates all data three times to 16 alternating backup servers' SSDs. After the logging process, the master server is killed and its data is recovered in parallel on 16 backup servers (called recovery masters in RAMCloud).

2) Results: In figure 6, recovery and replication is split for DXRAM, but not for RAMCloud as those two phases are indistinguishable here. In RAMCloud replayed objects are replicated during the recovery; in DXRAM after the recovery to reduce unavailability times. Therefore, Figure 6 also shows the re-replication times for DXRAM (gray boxes). Even when including this step (not necessary for immediate availability), DXRAM outperforms RAMCloud. With 1 MB objects DXRAM logs around 30% faster (bottlenecked by the network) and recovers around 140% faster. With smaller objects the edge grows even further. With 64 byte objects, DXRAM's recovery is more than 9 times faster than RAMCloud's (logging 3.5 times).

During the logging phase, RAMCloud creates all objects first and then replicates entire segments (8 MB) to speed-up this phase. DXRAM on the other side logs every single object (1 MB and 5 MB objects) or logs in batches of ten which is a more realistic behavior. Still, the logging phase is significantly shorter in DXRAM.

The recovery phase in RAMCloud differs from DXRAM: RAMCloud uses a log in RAM and distributes exact copies of the log segments to SSD on backup servers. As a consequence, during recovery, every object of the failed server could possibly be in every segment (in different versions as well). Thus, when recovery masters gather objects partition-wise, every single segment must have been read (in parallel) and all objects of all partitions must be sent over network to the right recovery master. Furthermore, during replay, every recovered object must be replicated three times as old backups are unusable. Those segments might contain objects of all partitions and not only the partition of one recovery master. Hence, in RAMCloud, every object is sent over the network four times during recovery whereas in our proposed approach only once.

The test with 5 MB objects could not be executed on RAMCloud as RAMCloud's maximum object size is 1 MB. However, this test shows the functionality of handling large chunks in DXRAM. The recovery is 0.5 sec slower than with 1 MB objects because for each object every chain link must be recovered before the object can be stored in memory which impairs the parallelism between Recovery-Main-Thread/Recovery-Helper-Threads and Recovery-Writer-Thread. Yet, the recovery is still under 2 sec.

C. Yahoo! Cloud Serving Benchmark

1) Description: The Yahoo! Cloud Serving Benchmark (YCSB) was designed to quantitatively compare distributed serving storage systems [21]. The benchmark offers a set of simple operations (reads, writes, range scans) and a tabular key-value data model to evaluate online storage systems regarding their elasticity, availability and replication. Furthermore, YCSB is easily extensible for new storage systems and new workloads. For our evaluation we used an individual workload typical for social media networks (for an evaluation of DXRAM's performance with other systems and more workloads refer to [11]): ten 64-byte objects per key, 15×10^6 keys per server, zipfian distribution, 90 % read and 10 % write operations, 10^8 operations (either reading all 10 objects per key or writing one object per key).

All storage servers are used as masters and backup servers. We use 48 storage servers, with a total of 7.2 billion 64-byte chunks in RAM and 21.6 billion log entries on SSD, and 24 YCSB clients for benchmarking. Each YCSB client is configured to emulate 100 clients using one thread per client resulting in 2,400 clients. During the benchmark phase, three masters are killed to analyze the recovery performance with high overall system load. Operations are never aborted but repeated until successful and the first backup peers are chosen disjunctive. Additionally, we use 7 superpeers to divide the load for lookup requests during recovery and to speed-up failure propagation (90/10 default ratio for peers/superpeers).

D. Results

Figure 7 shows the operation throughput, maximum and average response times of the first 300 seconds with 3 single server failures at second 24, 78 and 184. The benchmark was finished after additional 12 minutes and 42 seconds with no reportable incidents. The 24 clients executed around 2×10^6 operations per second resulting in 18×10^6 reads (10 chunks are read per read operation) and 2×10^5 writes every second. The average response time for completed remote operations (read or write) is around 1.3 ms. The maximum response time represents the slowest operation of all operations of all 24 clients during the last second. The average maximum response time is slightly above 100 ms and never over 500 ms in failure-free intervals.

The server failures impair the system for a short period, only (figure 7). The maximum response time of all clients throughout the whole benchmark is around 2.6 seconds, recorded during the first failure. The second and third failures were masked even faster. The recovery took 1.8, 1.3 and 1.6 seconds. The remaining time is spent for failure detection and ramping up of the clients. The evaluation with YCSB shows that DXRAM can quickly recover after consecutive single server failures even under heavy load, without disruptive interruption to the running application.

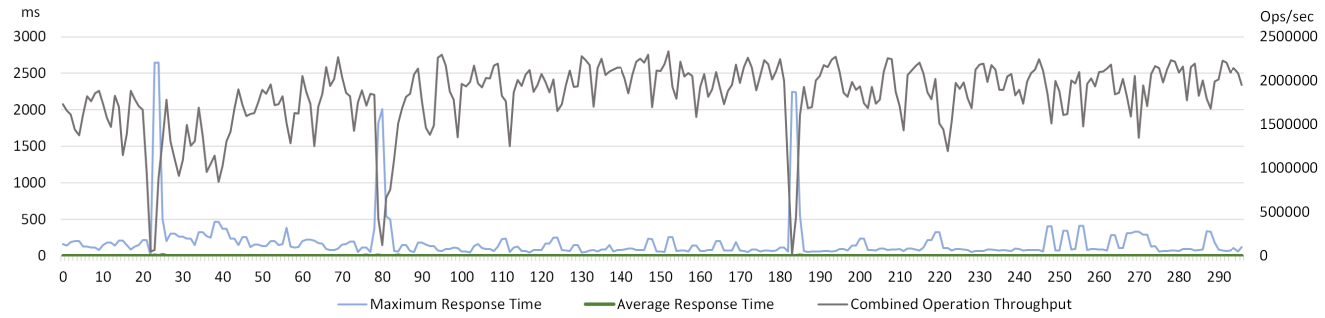

Figure 7: *YCSB - Maximum Client Request Times and Overall Throughput*

VIII. Conclusions

Low-latency data access is important for many application domains like online graph analytics or large-scale interactive applications. Distributed in-memory systems address this challenge, but data needs to be stored on persistent storage to allow masking server failures and power outages. Re-filling memory storage from traditional databases in case of failures is too slow and can result in long downtimes. Hence, parallel recovery approaches have been proposed (e.g. Google's Bigtable or RAMCloud). However, those systems have not been designed to efficiently handle small data objects found in many graph applications. In this paper we propose a fast parallel backup and recovery approach optimized for small data objects.

We propose a novel range-based backup replica management and highly parallel recovery strategy based on a two-level remote logging concept. The local recovery mechanisms are highly optimized for small data objects but also support larger objects. All concepts have been implemented in DXRAM (open source) but can be applied to other systems, too.

The evaluation results of experiments in the Microsoft's Azure cloud show that DXRAM is able to recovery a server storing 500 million 64-byte objects in under 2 sec; even under heavy load (generated by 2,400 clients emulated by the YCSB benchmark). And the recovery comparison with RAMCloud shows that DXRAM outperforms RAMCloud up to a factor of 9 for small data objects.

Future work includes optimizations and evaluation of DXRAM and the recovery using Infiniband. In addition, we have already implemented a basic graph processing framework on top of DXRAM and are adopting some bioinformatics graph applications which will allow further insights on the proposed concepts and likely introduce new challenges.

References

[1] R. Nishtala *et al.*, "Scaling memcache at facebook," in *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, Illinois, 2013.

[2] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," 2010.

[3] A. Gulli and A. Signorini, "The indexable web is more than 11.5 billion pages," in *Special interest tracks and posters of the 14th international conference on World Wide Web*, 2005.

[4] "Gemfire," http://www.vmware.com/products/vfabric-gemfire/overview.

[5] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, 2015.

[6] D. Ongaro *et al.*, "Fast crash recovery in ramcloud," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011.

[7] F. Klein and M. Schöttner, "Dxram: A persistent in-memory storage for billions of small objects," in *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2013 International Conference on*, 2013.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06, 2006.

[9] R. Stutsman, "Durability and crash recovery in distributed inmemory storage systems," dissertation, Stanford University, The Department of Computer Science, Stanford, CA, USA, 2013.

[10] F. Klein, K. Beineke, and M. Schöttner, "Memory management for billions of small objects in a distributed in-memory storage," in *IEEE Cluster 2014*, 2014.

[11] K. Beineke, S. Nothaas, and M. Schöttner, "High throughput log-based replication for many small in-memory objects," in *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[12] B. B. C. V. Srinivasan, "A real-time nosql db which preserves acid."

[13] S. Sanfilippo and P. Noordhuis, "Redis," 2009.

[14] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," ser. SOCC '14, 2014.

[15] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," ser. SIGCOMM '01, 2001.

[16] H. Lu, Y. Y. Ng, and Z. Tian, "T-tree or b-tree: main memory database index structure revisited," in *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, 2000.

[17] B. Atikoglu *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*, ser. SIGMETRICS '12, 2012.

[18] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer *et al.*, "Oceanstore: An architecture for global-scale persistent storage," *ACM Sigplan Notices*, 2000.

[20] F. Klein, K. Beineke, and M. Schöttner, "Distributed range-based meta-data management for an in-memory storage," in *LNCS Europar Workshop Proceedings, 4th Big Workshop on Big Data Managements in Clouds*, 2015.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

Chapter 6.

DXRAM's Fault-Tolerance Mechanisms Meet High Speed I/O Devices

This chapter summarizes the contributions and includes a copy of our report [12].

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "DXRAM's Fault-Tolerance Mechanisms Meet High Speed I/O Devices". In: ArXiv e-prints (July 2018). arXiv:1807.03562 [cs.DC]

6.1. Paper Summary

This report is based on [15] and [14]. We present optimizations of the logging and reorganization and introduce additional backup placement strategies. Furthermore, the logging implementation is described in greater detail with focus on direct disk access. To improve the logging performance, we introduced concepts of DXNet [13] to the logging architecture like lock-free queues and native memory access and also implemented direct access to the SSD bypassing the kernel page cache. The latter allows keeping a constant performance even if DXRAM data occupies most of the main memory. Furthermore, we refined the log selection strategy of the reorganization to increase the efficiency of the reorganization. At last, the replica placement is extended by an adapted copyset approach [24] to reduce the data loss probability without increasing the replication factor.

6.2. Importance and Impact on Thesis

In this report, we cover details of the logging and backup placement which we could not address in the publications [15] and [14] because of space constraints. Additionally, the author of this thesis applied concepts from DXNet to the logging of DXRAM and upgraded the reorganization and backup placement after the papers were published. Furthermore, this report concludes the thesis by bringing together the backup mechanisms with the high-speed networking of DXNet.

6.3. Personal Contribution

This publication includes three contributions: (1) backup placement strategies, (2) optimized logging architecture and (3) reorganization analysis. The backup placement strategies are inspired by [24]. The basic concept of copysets was applied and adjusted to DXRAM by Kevin Beineke, the author of this thesis, as well as, the optional locality-awareness and the disjunctive backup peer selection. The reorganization analysis includes the segment selection optimization based on timestamps. This idea was proposed by Rosenblum et al. in [97] and applied to DXRAM by Kevin Beineke.

The optimized logging architecture uses many ideas of DXNet like lock-free ring-buffers and zero-copy reading/writing from/to device. The concepts were applied by Kevin Beineke. The direct access to the disk was firstly introduced by Christian Gesse whose bachelor thesis [39] covers different disk access methods in Linux systems with focus on Solid State Drives. While the bachelor thesis, which was guided by Kevin Beineke, addresses the access methods very well, the conceptual formulation was limited and the result was not integrated into DXRAM's logging architecture. The author of this thesis integrated the work of Christian Gesse which required modifications to the logging architecture and developed further optimizations for the direct access to the disk.

The technical report was written and structured by Kevin Beineke. Prof. Dr. Michael Schöttner and Stefan Nothaas participated in discussions and reviewed the report.

DXRAM's Fault-Tolerance Mechanisms Meet High Speed I/O Devices

Kevin Beineke, Stefan Nothaas and Michael Schöttner

Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

***Abstract*—In-memory key-value stores provide consistent low-latency access to all objects which is important for interactive large-scale applications like social media networks or online graph analytics and also opens up new application areas. But, when storing the data in RAM on thousands of servers one has to consider server failures. Only a few in-memory key-value stores provide automatic online recovery of failed servers. The most prominent example of these systems is RAMCloud. Another system with sophisticated fault-tolerance mechanisms is DXRAM which is optimized for small data objects. In this report, we detail the remote replication process which is based on logs, investigate selection strategies for the reorganization of these logs and evaluate the reorganization performance for sequential, random, zipf and hot-and-cold distributions in DXRAM. This is also the first time DXRAM's backup system is evaluated with high speed I/O devices, specifically with 56 GBit/s InfiniBand interconnect and PCI-e SSDs. Furthermore, we discuss the copy-set replica distribution to reduce the probability for data loss and the adaptations to the original approach for DXRAM.**

***Keywords*—Reliability; Remote replication; Flash memory; InfiniBand; Java; Data centers; Cloud computing;**

I. Introduction

In [1] and [2], we described the distributed logging and highly parallelized recovery approaches of DXRAM. While we demonstrated that DXRAM outperforms state-of-the-art systems like RAMCloud, Aerospike or Redis on typical cluster hardware, we could not explore the limits of DXRAM's logging approach because of hardware limitations. In this report, we evaluate the backup system of DXRAM with fast hardware and present three different optimizations: (1) an improved pipeline from network to disk on backup servers, (2) a new segment selection strategy for the reorganization of logs and (3) an adapted copy-set approach to decrease the probability for data loss.

The evaluation shows that DXRAM is able to log more than 4,000,000 64-byte chunks per second received over an InfiniBand network. Larger chunks, e.g., 512-byte chunks, can be logged at nearly 1 GB/s, saturating the PCI-e SSD. The reorganization is able to keep the utilization under 80% most times for all update distributions (sequential, random, zipf and hotNcold) while maintaining a high logging throughput. Furthermore, we show that the two-level logging concept improves the performance up to more than nine times.

The structure of this report is as follows. Section II outlines the basic architecture of DXRAM. In Section III, we depict the related work on logging. In Section IV, we give a top-down overview of DXRAM's logging followed by a detailed description of the logging pipeline. Section VII discussed the related work regarding the log reorganization and segment selection. DXRAM's reorganization approach is presented in Section VIII. The related work for backup distribution is outlined in Section IX which is followed by the modified copyset approach of DXRAM in Section X. In Section XI, we evaluate the proposed concepts of Sections V and VIII. Section XII concludes this report.

II. DXRAM

DXRAM is an open source distributed in-memory system with a layered architecture, written in Java [3]. It is extensible with additional services and data models beyond the key-value foundation of the DXRAM core. In DXRAM, an in-memory data object is called a *chunk* whereas an object stored in a log on disk is referred to as *log entry*. The term *disk* is used for Solid-State Drives (SSD) and Hard Disk Drives (HDD), interchangeably.

A. Global Meta-Data Management

In DXRAM, every server is either a peer or a superpeer. Peers store chunks, may run computations and exchange data directly with other peers, and also serve client requests when DXRAM is used as a back-end storage. Peers can be storage *servers* (with in-memory chunks), *backup servers* (with logged chunks on disk) or both. Superpeers store global meta-data like the locations of chunks, implement a monitoring facility, detect failures and coordinate the recovery of failed peers, and also provide a naming service. The superpeers are arranged in a zero-hop overlay which is based on Chord [4] adapted to the conditions in a data center. Moreover, every peer is assigned to one superpeer which is responsible for meta-data management and recovery coordination of its associated peers. During server startup, every server receives a unique node ID.

Every superpeer replicates its data on three succeeding superpeers in the overlay. If a superpeer becomes unavailable, the first successor will automatically take place and stabilize the overlay. In case of a power outage, the meta-data can be reconstructed based on the recovered peers' data. Thus, storing the meta-data on disk on superpeers is not necessary.

Every chunk in DXRAM has a 64-bit globally unique chunk ID (CID). This ID consists of two separate parts: a 16-bit node ID of the chunk's creator and a 48-bit locally

unique sequential number. With the creator's node ID being part of a CID, every chunk's initial location is known a-priori. But, the location of a chunk may change over time in case of load balancing decisions or when a server fails permanently. Superpeers use a modified B-tree [5], called *lookup tree*, allowing a space efficient and fast server lookup while supporting chunk migrations. Space efficiency is achieved by a per-server sequential ID generation and ID re-usage in case of chunk removals allowing to manage chunk locations using CID ranges with one entry for a set of chunks. In turn, a chunk location lookup will reply with a range of CIDs, not a single location, only. This reduces the number of location lookup requests. For caching of lookup locations on peers, a similar tree is used further reducing network load for lookups.

B. Memory Management

The sequential order of CIDs (as described in section II-A) allows us to use compact paging-like address translation tables on servers with a constant lookup time complexity. Although, this table structure has similarities with well known operating systems' paging tables we apply it in a different manner. On each DXRAM server, we use the lower part (LID) of the CID as a key to lookup the virtual memory address of the stored chunk data. The LID is split into multiple parts (e.g., four parts of 12 bit each) representing the distinct levels of the paging hierarchy. This allows us to allocate and free page tables on demand reducing the overall memory consumption of the local meta-data management. Complemented with an additional level indexed by node ID storing of migrated chunks is possible as well. DXRAM uses a tailored memory allocator with very low footprint working on a large pre-reserved memory block. For performance and space efficiency reasons, all memory operations are implemented using the Java Unsafe class.

Chunks store binary data and each chunk ID (CID) contains the *creator*. Chunks can be migrated to other servers for load balancing reasons. Migrated chunks are then called *immigrated chunks* on the receiver and *emigrated chunks* on the creator. Finally, there are *recovered chunks* stored on a new *owner* after a server failure.

III. Related Work on Logging

Numerous distributed in-memory systems have been proposed to provide low-latency data access for online queries and analytics for various graph applications. These systems often need to aggregate many nodes to provide enough RAM capacity for the exploding data volumes which in turn results in a high probability of node failures. The latter includes soft- and hardware failures as well as power outages which need to be addressed by replication mechanisms and logging concepts storing data on secondary storage. Because of space constraints, we can only discuss the most relevant work.

RAMCloud is an in-memory system, sharing several objectives with DXRAM while having a different architecture, providing a table-based in-memory storage to keep all data always in memory. However, the table-based data model of RAMCloud is designed for larger objects and suffers from a comparable large overhead for small data objects [6]. It uses a distributed hash table, maintained by a central coordinator, to map 64-bit global IDs to nodes which can also be cached by clients. DXRAM on the other hand uses a superpeer overlay with a more space-efficient range-based meta-data management. For persistence and fault tolerance it implements a log-based replication of data on remote nodes' disks [7]. In contrast to other in-memory systems, RAMCloud organizes in-memory data also as a log which is scattered for replication purposes across many nodes' disks in a master slave coupling [8]. Scattering the state of one node's log on many backup nodes allows a fast recovery of 32 GB of data and more. Obviously, logging throughput depends on the I/O bandwidth of disks as well as on the available network bandwidth and CPU resources for data processing. RAMCloud uses a centralized log-reorganization approach executed on the in-memory log of the server which resends re-organized segments (8 MB size) of the log over the network to backup nodes. As a result, remaining valid objects will be re-replicated over the network after every reorganization iteration to clean-up the persistent logs on remote nodes. This approach relieves remote disks but at the same time burdens the master and the network. DXRAM uses an orthogonal approach by doing the reorganization of logs on backup nodes avoiding network traffic for reorganization. Furthermore, DXRAM does not organize the in-memory storage as a log but uses updates in-place. Finally, RAMCloud is written in C++ and provides client bindings for C, C++, Java and Python [7] whereas DXRAM is written in Java.

Aerospike is a distributed database platform providing consistency, reliability, self-management and high performance clustering [9]. Aerospike uses Paxos consensus for node joining and failing and balances the load with migrations. In comparison, DXRAM also offers a migration mechanism for load balancing. The object lookup is provided by a distributed hash table in Aerospike. Aerospike is optimized for TCP/IP. Additionally, Aerospike enables different storage modes for every namespace. For instance, all data can be stored on SSD with indexes in RAM or all data can be stored in RAM and optionally on SSD with a configurable replication factor. As Aerospike is a commercial product, not many implementation details are published except that it internally writes all data into logs stored in larger bins optimized for flash memory. The basic server code of Aerospike is written in C and available clients include bindings for C, C#, Java, Go, Python, Perl and many more.

Redis is another distributed in-memory system which can be used as an in-memory database or as a cache [10]. Redis provides a master-slave asynchronous replication and different on-disk persistence modes. To replicate in-memory objects, exact copies of masters, called slaves, are filled with all objects asynchronously. To overcome power outages and node failures, snapshotting and append-only logging with periodical rewriting can be used. However, to replicate on disk the node must also be replicated in RAM which increases the total amount of RAM needed drastically. This is an expensive approach and very different from the one of DXRAM where remote replicas are stored on SSD only. Obviously, Redis has no problems with I/O bandwidth as it stores all data in RAM on slaves and can postpone flushing on disk as needed. Furthermore, reorganization is also quite radical compared to DXRAM as Redis just reads in a full log to compress it which is of course fast but introduces again a lot of RAM overhead.

Redis is written in C and offers clients for many programming languages like C, C++, C#, Java and Go.

Log-structured File Systems are an important inspiration for the log-based replication of RAMCloud and DXRAM. A log is the preferred data structure for replication on disk as a log has a superior write throughput due to appending objects, only. But, a log requires a periodical reorganization to discard outdated or deleted objects in order to free space for further write accesses. In [11] Rosenblum and Ousterhout describe a file system which is based on a log. Furthermore, a cleaning policy is introduced which divides the log into segments and selects the segment with best cost-benefit ratio for reorganization. DXRAM divides a log into segments as well. However, due to memory constrains the cost-benefit formula is limited to the age and utilization of a segment (more in section VII).

Journaling is used in several file systems to reconstruct corruptions after a disk or system failure. A journal is a log that is filled with meta-data (and sometimes data) before committing to main file system. The advantage is an increased performance while writing to the log as appending to a log is faster than updating in-place but requires a second write access. The to be described two-level logging of DXRAM also uses an additional log to efficiently utilize an SSD. In contrast to journaling, we use this log only for small write accesses from many remote nodes to allow bulk writes without impeding persistence.

IV. Logging in DXRAM - An Overview

In this section, we describe the basic logging architecture of DXRAM which is subject of [1]. Below, we distinguish two different roles: *Masters* are DXRAM peers, store chunks (see Section II) and replicate them on *backup servers*. A backup server might also be a master and vice versa.

Replicating multi-billion small data objects in RAM is too expensive and does not allow to mask power outages. Therefore the backup data structures of DXRAM are designed to maximize throughput of SSDs by using logs.

1) Two-Level Logging: We divide every server's data into backup zones of equal size. Each backup zone is stored in one separate log on every assigned backup server (typically three per backup zone). Those logs are called *secondary logs* and are the final destination for every replica and the only data structure used to recover data from. By sorting backups per backup zone, we can speed-up the recovery process by avoiding to analyze a single log with billions of entries mixed from several masters (as required RAMCloud). The two-level log organization also ensures that infrequent written secondary logs do not thwart highly burdened secondary logs by writing small data to disk and thus utilizing the disk inefficiently. At the same time, incoming objects are quickly stored on disk to sustain power outages.

First, every object received for backup is written to a ring buffer, called *write buffer*, to bundle small request (Figure 1). This buffer is a lock-free ring-buffer which allows concurrently writing into the buffer while it is (partly) flushed to disk. During the flushing process, which is triggered periodically or if a threshold is reached, the content is sorted by backup zones

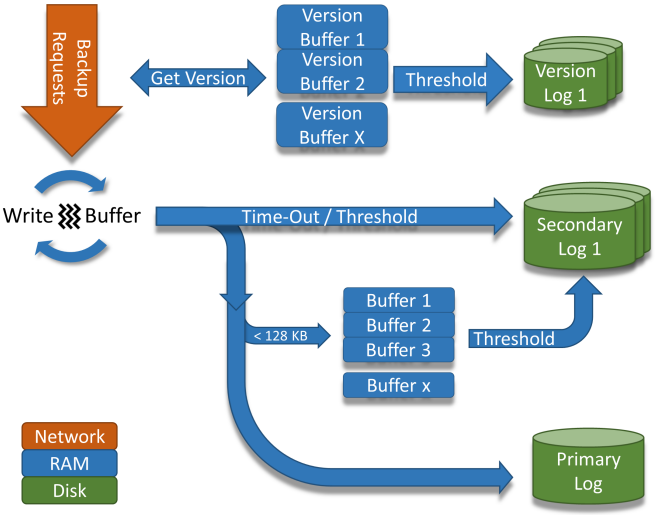

Figure 1: *The Logging architecture. Every object is buffered first. Depending on the amount of data per backup zone, the objects are either directly written to their associated secondary log or to both primary log and secondary log once there is enough data. Versions are determined by inquiring the corresponding version buffer, which is flushed to its version log frequently.*

to form larger batches of data in order to allow bulk writes to disk. If one of those batches is larger than a predefined threshold (e.g., 32 flash pages of the disk), it is written directly to the corresponding secondary log.

In addition to the secondary logs, there is **one** *primary log* for temporarily storing smaller batches of **all** backup zones to guarantee fast persistence without decreasing disk throughput. The smaller batches are also buffered in RAM separately, in so called *secondary log buffers*, for every secondary log and will eventually be written to the corresponding secondary log when aggregated to larger batches. Obviously, with this approach some objects will be written to disk twice but this is outweighed by utilizing the disk more efficiently. Waiting individually for every secondary log until the threshold is reached without writing to the primary log, on the other hand, is no option as the data is prone to get lost in case of a power outage.

2) Backup-side Version Control: Masters do not store version information in RAM. Versions are necessary for identifying outdated data in the logs, so the backup servers employ a version control used for the reorganization and recovery. A naïve solution would be to manage every object's version in RAM on backup servers. Unfortunately, this approach consumes too much memory, e.g., at least 12 bytes (8-byte CID and 4-byte version) for every object stored in a log easily sums up to many GB in RAM which is not affordable. Storing the entire version information on disk is also not practical because of performance reasons as this would require reads for each log write. Caching recent versions in memory could possibly help for some access patterns but for the targeted application domain would either cause many read accesses for cache misses or occupy a lot of memory. Instead, we propose a version manager which runs on every backup server and utilizes one *version buffer* per secondary log. The version buffer holds recent versions for this secondary log in RAM

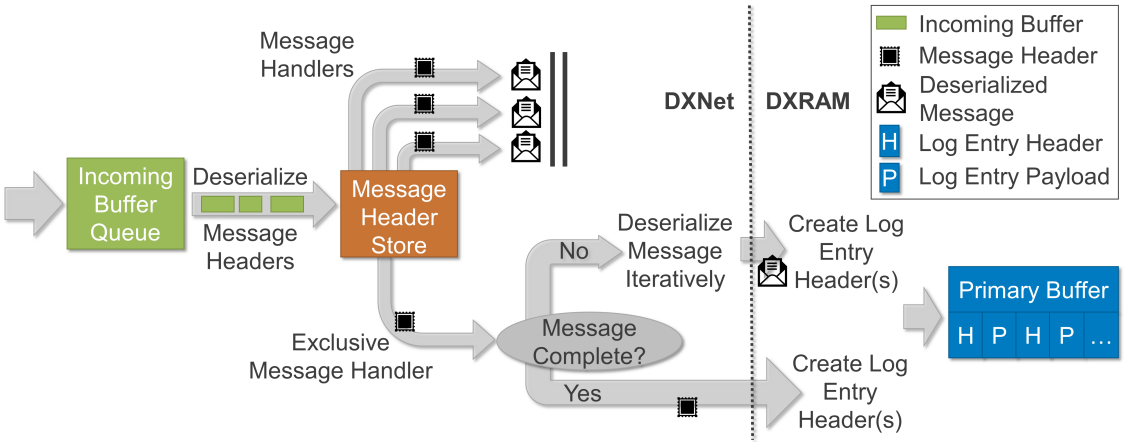

Figure 2: *From Network to Write Buffer*

until it is flushed to disk. In contrary to a simple cache solution, DXRAM's version manager avoids loading missing entries from secondary storage by distinguishing time spans, called *epochs*, which serve as an extension of a plain version number. At the beginning of an epoch, the version buffer is empty. If a backup arrives within this epoch, its CID will be added to the corresponding version buffer with version number 0. Another backup for the same object within this epoch will increment the version number to 1, the next to 2 and so on. When the version buffer is flushed to disk, all version information is complemented by the current epoch, together creating a unique version. In the next epoch the version buffer is empty again. An epoch ends when the version buffer reaches a predefined threshold allowing to limit the buffer size, e.g., 1 MB per log. During flushing to disk, a version buffer is compacted resulting in a sequence of (CID, epoch, version)-tuples with no ordering. This sequence is appended to a file on disk, creating a log of unique versions for every single secondary log. We call it a *version log*. Over time, a version log contains several invalid entries which are tuples with outdated versions. To prevent a version log from continuously growing, it is compacted during reorganization.

V. Logging in DXRAM - From Network to Disk

In this section, we present all stages involved on a backup server from receiving a chunk over a network connection to writing the chunk to disk. This includes the deserialization of the message object (in Section V-A), the creation of a log entry header to identify a chunk within a log (in Section V-B) and the aggregation of all chunks of all backup zones in the **write buffer** (in Section V-C). Furthermore, this section covers the sorting and processing of the write buffer to create large batches which can be written to disk efficiently (in Section V-D). After that, we briefly describe all data structures on disk and how they are accessed (in Section V-E). Finally, we discuss different disk access methods and describe all three implemented methods thoroughly (in Section V-F).

A. Message Receipt and Deserialization

For sending replicas, DXRAM uses DXNet, a network messaging framework which utilizes different network technologies, currently supporting Ethernet and InfiniBand. DXNet guarantees packet and message ordering by using a special network handler, which is used for logging. DXRAM defines a fixed replication ordering for every backup zone enabling the application of asynchronous messages for chunk replication. Server failures are handled by re-replicating the chunks to another backup server and adjusting the replication ordering (the failed server is removed and the new backup server is added at the end).

There are two major messages involved in the logging process. One for replicating one or multiple chunks to a specific backup zone (a log message) and one for creating a new backup zone on a backup server which includes creating the secondary and version log and their corresponding buffers. All chunks of one log message belong to the same backup zone (allocation is performed on masters). Therefore, the range ID (identifier for a backup zone which is also called backup range) and the owner is included in the message once, only, followed by the chunk ID, payload length and payload of the first chunk. Typically, messages are created and deserialized entirely by DXNet, i.e., a new message object is created and all chunks are deserialized (in this case into a ByteBuffer) to be processed (logged) by the message handler. For the logging, we optimized this step by deserializing directly into the **write buffer** (see Section IV-1) to avoid creating a message object (allocations are rather expensive) and copying from the deserialized ByteBuffer into the write buffer. Whenever a message is contained entirely in the received incoming buffer, the log message is deserialized into the write buffer. If not, i.e., the log message is split into at least two buffers, we delegate the deserialization to DXNet in order to reduce complexity (see Figure 2). The performance is mostly unaffected by the latter, as log message splitting is rather seldom. The detection is done within DXNet: if the message size is smaller than the number of remaining bytes in the buffer, a special message receiver is called which is registered by the logging component on system initialization. Otherwise, a normal message receiver is called. The difference is that a normal message receiver operates on a deserialized message object and the special message receiver on a message header and not yet deserialized ByteBuffer. We hide the complexity of the deserialization in the special message receiver by using DXNet's Importer which offers deserialization methods for primitives, arrays and objects.

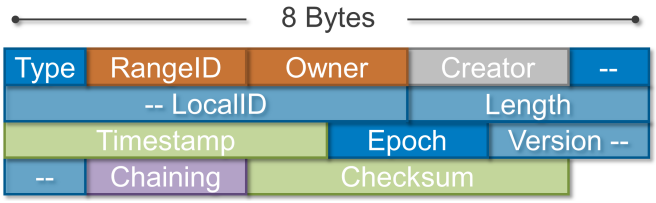

Figure 3: *Log Entry Header. Orange: for write buffer and primary log, only. Grey: for migrated or recovered chunks. Green: optional/configurable. Purple: for chunks larger than 4 MB. Dark blue: mandatory, minimal size. Transparent blue: maximum size*

B. Log Entry Headers

We cannot simply copy a chunk (or a batch of chunks) from message buffer to write buffer as every log entry consists of a log entry header followed by the payload. The log entry header has to be created just before writing the entry to the write buffer as it contains a unique version number which has to be determined by inquiring the version buffer of given backup zone. Optionally, the log entry header contains a timestamp and a CRC checksum which have to be recorded/generated as well. Figure 3 shows all fields of a log entry header.

- Type: this field specifies the type of the log entry header and stores the sizes of the LocalID, length, version and chaining fields. There are three different types of log entry headers: (1) a `DefaultSecLogEntryHeader` which is used for chunks stored in a secondary log. (2) A `MigrationSecLogEntryHeader` for migrated/recovered chunks stored in a secondary log and (3) a `PrimLogEntryHeader` for log entries stored in write buffer or primary log. In write buffer, every log entry is preceded by a `PrimLogEntryHeader`. When the log entry is written to primary log, the header remains unchanged. For writing the log entry to secondary log (or secondary log buffer) the `PrimLogEntryHeader` is converted into a `Default-` or `MigrationSecLogEntryHeader` by removing the RangeID and owner fields (both specified by the secondary log the entry is stored in). For converting to a `DefaultSecLogEntryHeader`, the creator is removed as well (the creator is the same as the creator of the backup zone).
- RangeID and Owner: for log entries stored in write buffer and primary log to identify the backup zone the log entry belongs to.
- Creator: if the creator differs from the creator of the backup zone (if migrated or recovered), it has to be stored in order to restore the CID during recovery.
- LocalID: to identify a chunk within a backup zone. Can be one, two, four or six bytes (defined in type field).
- Length: the payload size. Can be zero, one, two or three bytes. Maximum size for a log entry is 4 MB (half the size of a segment which is configurable). Larger chunks are split to several log entries (see Chaining).

```
uint32_t i = 0;
while (i + 8 <= length) {
  crc = _mm_crc32_u64(crc, *((uint64_t *)
      &data[i + offset]));
  i += 8;
}

if (i + 4 <= length) {
  crc = _mm_crc32_u32(crc, *((uint32_t *)
      &data[i + offset]));
  i += 4;
}

if (i + 2 <= length) {
  crc = _mm_crc32_u16(crc, *((uint16_t *)
      &data[i + offset]));
  i += 2;
}

if (i < length) {
  crc = _mm_crc32_u8(crc, data[i +
      offset]);
  i++;
}
```

Figure 4: *Fast Checksum Computation*

- Timestamp: the timestamp represents the point in time the log entry was created. More precisely, the seconds elapsed since the log component was created. The timestamp is optional and used for the optimized segment selection of the reorganization (see Section VIII).
- Epoch and Version: together epoch and version describe a unique version number for given CID. The version field can be zero, one, two or four bytes. The most used version 1 takes no space to store.
- Chaining: not available for chunks smaller than 4 MB. Otherwise, the first byte represents the position in the chain and the second byte the length of the chain. Theoretical maximum size for chunks with default configuration: $256 * 4\text{MB} = 1\text{GB}$. The segment size can be increased to 16 MB to enable logging of 2 GB chunks which is the maximum size supported by DXRAM.
- Checksum: the CRC32 checksum is used to check for errors in the payload of a log entry during the recovery. If available, the checksum is generated using the SSE4.2 instructions of the CPU (see Figure 4).

C. Write Buffer

The write buffer is a ring-buffer which is accessed by a single producer, the exclusive message handler (see Figure 2), and a single consumer, the *BufferProcessingThread*. Every chunk to be logged is written to the write buffer first. Beneath the data in the write buffer, we also store backup zone specific information in a hash table. The keys for the hash table are the owner and range ID (combined to an integer) of the backup zone. The value is the length of all current log entries belonging to the backup zone. We use a custom-made hash table based on linear probing as it is faster than Java's

hashmap and avoids allocations due to reusing the complete data structure for the next iteration. Access to the hash table is not atomic but must be in sync with the write buffer. Thus, updating the write buffers positions and the hash table is locked by a spin lock, even though the write buffer itself could be accessed lock-free. However, the length information is important to distribute the log entries to segments (see Section V-D). We do not wait in case the lock cannot be acquired but try again directly because the critical areas are small as well as the collision probability (*BufferProcessingThread* is in critical area for a short time every 16 MB or 100 ms, see Section V-D).

If the write buffer is full, the exclusive message handler has to wait for the *BufferProcessingThread* to flush the write buffer. We use LockSupport.parkNanos(100) which is a good compromise between reducing the CPU load while waiting and being responsive enough. When writing to the write buffer, the overflow needs to be dealt with by continuing at the buffers start position.

D. Flushing and Sorting

The *BufferProcessingThread* flushes the write buffer periodically (every 100 ms) and based on a threshold (half the size of the write buffer; default: 32 MB). The flushing can be done concurrently to further filling the write buffer once the metadata (front and back pointer and the hash table) has been read and set accordingly. Thus under load, half of the write buffer is written to disk while the other half is filled enabling a constant utilization.

Priority Flush: The flushing can also be triggered by the recovery and reorganization to ensure all relevant data is stored in the corresponding secondary logs. Additionally, whenever a version buffer is full, a priority flush is triggered to flush the version buffer consequently.

The flushing process does not simply write the data as it is to the corresponding secondary logs, but sorts the log entries by backup zone to create the largest possible batches which can be written efficiently to disk. First, we use the information about the total length of a backup zone's data (stored in the hash table) to supply ByteBuffers to store the sorted data. None of the buffers exceeds the size of a segment as we want to write the buffer's content with one write access, if possible. For example, 14 MB of data belonging to one backup zone might be split to two 8 MB buffers. It could also be split to one 8 MB and six 1 MB buffers depending on the available buffers in the buffer pool (described below). All buffers of one backup zone are collected in a Java object which is registered with an identifier for the backup zone (combined range ID and owner) in a hash table similar to the one for recording the lengths. This enables a fast lookup during the sorting process. The ordering within a backup zone is preserved because we iterate the write buffer from back pointer to front pointer and copy the log entries to the corresponding buffers. We do not fill previous buffers to reduce fragmentation, either, because of the ordering (a smaller succeeding log entry might fit in the previous buffer). Again, when copying the log entries the overflow of the write buffer must be considered. Additionally, the log entry headers are truncated when written to the buffers (see Figure 3).

1) Buffer Pool: To avoid constantly allocating new buffers when sorting the data, we employ a buffer pool which stores buffers in three configurable sizes (e.g., 64 x 0.5 MB, 32 x 1 MB and 8 x 8 MB for 8 MB segments) to support different access patterns. The buffer pool consists of three lock-free multi producer, single consumer ring buffers and buffers are chosen with a best-fit strategy. If all ring buffers run dry, the *BufferProcessingThread* waits for the next buffer being returned. Buffers are returned after they have been written to disk.

The buffers of all backup zones with less than 128 KB (default value) of data are merged and written to primary log. Additionally, the buffers are copied to the corresponding secondary log buffers (with further truncated headers) to enable fast flushing once a buffer is full. For backup zones with more than 128 KB of data, the buffer is directly written to secondary log. It might be necessary to flush the secondary log buffer first (can be merged with buffer if both together are not larger than a segment). The *BufferProcessingThread* does not execute the write accesses to disk, but registers the write access in a lock-free ring-buffer, called *WriterJobQueue*, to allow concurrent sorting/processing of new data while the data is written to disk (very important for synchronous access). The WriterJobQueue is synchronized by using memory fences, only. The jobs are pulled and executed by a dedicated writer thread.

After the flushing, the hash table is cleared and the back pointer of the primary buffer is set to previous front pointer.

E. Data Structures on Disk

Everything DXRAM's backup system writes to disk is arranged in logs. The primary log and secondary logs store replicas, the version logs version information for all logged chunks.

1) Primary Log: The primary log is used to ensure fast persistence for arbitrary access while efficiently utilizing the disk, i.e., if the write buffer stores log entries of many backup zones, all batches may be small and writing them to disk would slow down the disk considerably. Thus, all data is written to primary log with one large access and buffered in corresponding secondary log buffers. If a secondary log buffer is large enough to be written to disk efficiently (default: 128 KB), it will be flushed to secondary log.

The primary log is filled sequentially from beginning (position 0) to the end. It does not get reorganized or compacted in any way, nor is it used to recover from during the online recovery. Its only purpose, is to store small batches persistently to be recovered in case of a power outage, i.e., all servers responsible for at least one backup zone break down and not all secondary log buffers could be flushed prior to the failure.

If the primary log is full, all secondary log buffers are flushed and the position is set to 0. As secondary log buffers are flushed frequently the amount of data to be flushed in this scenario is rather small. Even in worst case scenario, only 128 KB per backup zone needs to be written to disk.

For the two-level logging, we assume the cluster servers do not have non-volatile random access memory (NVRAM) or battery backup. If they utilize NVRAM (with NVRAM we refer to byte-addressable non-volatile memory on main

memory layer, not flash memory used in SSDs), logging to disk is still necessary as replicating in NVRAM is too expensive and failed servers may be irreparable. However, the two-level logging is redundant as the write buffer and secondary log buffers can be accessed after rebooting (not implemented). For battery backed-up servers, the primary log is expendable, as well, because all secondary log buffers can be flushed while the server runs on battery (soft shut-down). We provide two options to optimize the logging for NVRAM and battery backup: (1) the threshold to decide whether the data is written to primary log or secondary log can be set to 0 or (2) the two-level logging can be disabled explicitly. In the first case, all aggregated batches are written directly to secondary log regardless of the size of the bulk. The second option disables the primary log, as well, but still utilizes secondary log buffers if batches are small.

2) Secondary Logs: Secondary logs eventually store all log entries on disk and are used for the recovery (online and global shut-down recovery). Secondary logs are subdivided into segments (default size: 8 MB) which is beneficial for the recovery and reorganization to limit the memory consumption by processing the log segment by segment. Furthermore, the segmentation allows reorganizing the parts of the log which are more likely outdated (see Section VIII). For writing new log entries to the secondary log, the segment boundaries must be respected because log entries must not span over two segments which would add unnecessary complexity to the recovery and reorganization.

As described before, log entries are sorted and copied into buffers with maximum size equal to the segment size. Usually, we write an entire buffer into one segment. When the buffer stores at least 6 MB (75% of the segment size), we open a new segment and write the buffer to the beginning of the new segment. If not (< 6 MB), we search for a used segment with enough space to write the entire buffer into. When none of the used segments have enough space to hold the buffer, we open a new segment, too. If all segments of a log are already in use, we split the buffer and gradually fill the segments with most free space. This is a compromise between maximum throughput while writing to the log (minimum write accesses, page-aligned access) and maximum efficiency of the reorganization (high utilization of the segments). If the buffer contains more data than there is free space in the secondary log, a high-priority request is registered for reorganizing the complete secondary log and the writer thread waits until the request was handled. If the secondary log's utilization breaches a configurable threshold (e.g., 85%) after writing to disk, a low-priority reorganization request is registered and the writer thread proceeds.

The presented writing scheme is used whenever the secondary log is not accessed by the reorganization thread. If the reorganization is in progress for the secondary log to write to, an **active segment** is chosen which can be filled concurrently to other segments being reorganized as it is locked to the reorganization. The active segment is exchanged if it is full (next log entry does not fit), only. Obviously, the currently reorganized segment cannot be used as an active segment. All other segments are free to be chosen. Furthermore, during concurrent access all write accesses to disk have to be serialized to avoid corrupting the file.

Secondary logs are recovered entirely by reading all log entries of the log (segment by segment) and storing the valid (and error-free) log entries in DXRAM's memory management. In order to keep recovery times low and to avoid secondary logs completely filling up, secondary logs have to be reorganized from time to time (see Section VIII). During the recovery, the reorganization is completely locked to avoid inconsistencies and to allocate all available resources to the recovery.

3) Version Logs: Version logs store the version numbers of log entries belonging to the same secondary log. Every version log is supported by a version buffer which holds all current versions of the latest epoch. At the end of every epoch, the version buffer is flushed to version log by appending all version numbers of the version buffer to the end of the version log. An epoch transition is initiated whenever the writer thread writes to a secondary log and the version buffer breached its flushing threshold (e.g., 65%). If the secondary log is reorganized simultaneously, the version buffer is not flushed but a low-priority reorganization request is registered. Prior to the actual reorganization, the reorganization thread has to read the entire version log and store all version numbers in a hash table which is used to validate the log entries during the reorganization. Then, the hash table is complemented by all version numbers currently stored in the version buffer and the epoch number is incremented. We exploit the situation to compact the version log by writing all version numbers stored in the hash table to the beginning of the version log. This way, we keep the version log small without a dedicated reorganization.

In case of a power failure, a version log might be ahead of its secondary log, i.e., a new version number might have been stored in version log whose corresponding log entry have not been written to secondary log prior to the hard shutdown. Therefore, it is important to not use the version log after a power failure to identify the most recent version of a chunk as this could result in not recovering a chunk at all by rejecting the most recent version in secondary log (which has a lower version number than registered in version log). Instead, we cache all chunks from all segments (in a hash table) and overwrite an entry if the version number is higher. The version log is used to determine deleted chunks, only. For the crash recovery of a single server (or multiple servers with at least one alive replica of every backup zone) and the reorganization, the versions are gathered, first. Then, the version log as well as the primary and secondary logs are flushed, prior to recovering/reorganizing the secondary log. Therefore, the version information cannot be more recent than the data. However, the opposite is possible (data newer than version). We solve this by considering all brand-new log entries (logged during the recovery/reorganization) to be valid, i.e., log entries created in the current epoch are kept (the epoch is not incremented during the recovery/reorganization after reading the versions).

F. Access Modes for Writing to Logs

DXRAM supports three different disk access modes to write to logs (primary, secondary and version logs): (1) writing to a RandomAccessFile, (2) writing to a file opened with O_DIRECT and SYNC flags and (3) writing to a RAW partition. The RandomAccessFile requires a byte array stored in Java heap to read and write to disk. The other

two access modes operate on page-aligned native arrays. In order to support both, we use ByteBuffers throughout the entire logging module. The ByteBuffers used for the RandomAccessFile are allocated in Java heap which allows accessing the underlying byte array used to read/write to disk. The ByteBuffers for direct access are stored in native memory by using the method `allocateDirect`. The access to the underlying byte array in native memory is done in a Java Native Interface (JNI) module by accessing it directly by address. The address is determined by using the call `Buffer.class.getDeclaredField("address")`. To avoid calling the reflecting method every time the buffer is accessed in the JNI module, we determine the address once and store it alongside the reference of the ByteBuffer in a wrapper which is used throughout the logging module. A performance comparison between Direct-, HeapByteBuffers and arrays can be found in Section XI-A.

Most of the ByteBuffers used to write to or read from disk are pooled to relief the garbage collection and speed-up the processing. The only two exceptions are the buffers used to write to primary log (length of all log entries to write to primary log differs significantly from write to write) and to flush the secondary log buffer if the new bulk to write exceeds the secondary log buffer size (e.g., 100 KB in secondary log buffer and 150 KB in write buffer).

During fault-free execution the current position within a log/segment and the length of a log/segment is stored in RAM (for performance reasons). For the recovery of a failed master, the information, stored on backup servers, is used as well. However, in case of a power failure the lengths and positions (irrelevant for the recovery) are unavailable. We cannot store the lengths on disk because this is too slow. Instead, every log's file is initialized with zeros and every write access to primary and secondary logs is followed by a 0 to mark the end. Whenever a log entry is read which starts with a 0, we know that the end of the segment/log is reached as the type field of a log entry header cannot be 0. And we do not have to mark the end of the version logs as the files are truncated after every write.

1) RandomAccessFile: The RandomAccessFile is probably the easiest and most comfortable way for random writes and reads to/from a file in Java. The RandomAccessFile is based on Java's `FileInput-` and `FileOutputStream` which use the read and write function of the operating system. In Linux all write and read accesses are buffered by the page cache (if the file was not opened with O_DIRECT flag), i.e., when writing to disk the buffer is first written to the page cache and eventually to disk (may be cached on disk as well). We discuss the dis-/advantages of the page cache later.

We create one file for every log (primary, secondary and version logs) in the file system (e.g., ext4). The files are opened in read-write mode ("rw"). Before every read/write access we seek to the position in file. The offset in the byte array can be passed to the read/write method.

2) Direct Access: To directly access a file, we have to use Linux functions, which cannot be accessed in Java. We use JNI to integrate a *C* program which handles the low-level access to files. Files are opened with `open`, read with `pread` and written with `pwrite`. The Linux kernel functions `pread` and

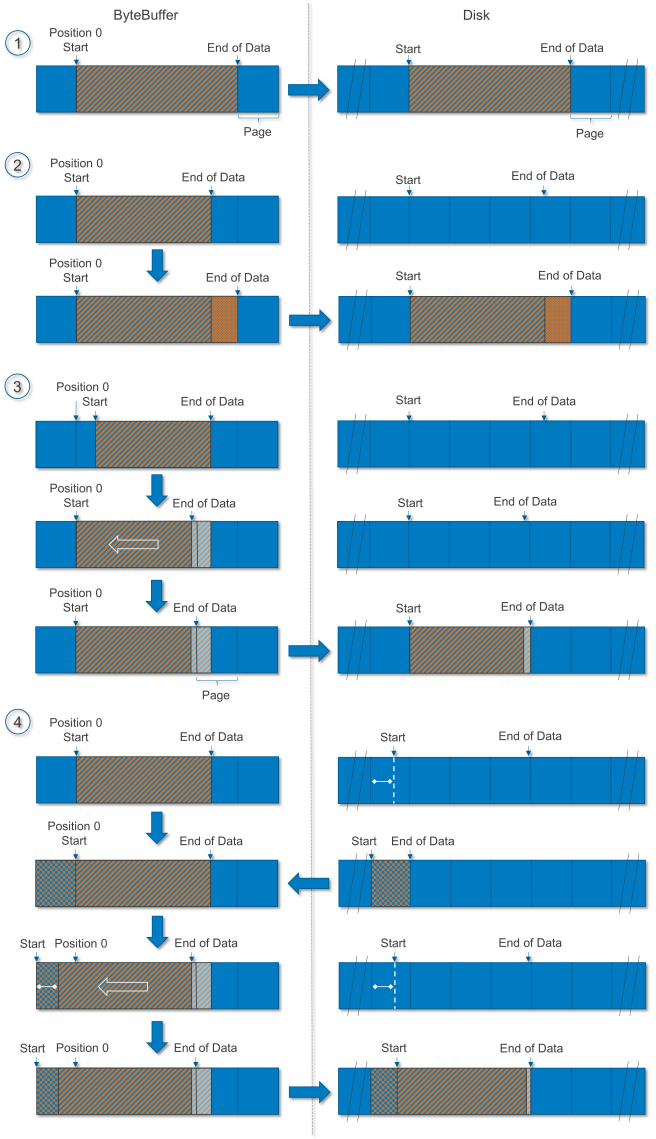

Figure 5: *Buffer and File Alignment*

`pwrite` have the same behavior as `read` and `write` but do not change the file pointer. The buffers' addresses are passed as longs and the file IDs as ints and are both managed in the Java part. Accessing files directly, without page-cache, requires the files to be opened with the O_DIRECT flag. We open logs with the following flags:

- O_CREAT: create the file if it not already exists
- O_RDWR: this file is going to be read and written
- O_DSYNC: write accesses return after the data was written (might be in disk cache). Metadata (e.g., timestamps) might not be updated yet
- O_DIRECT: this file is accessed directly without page-cache

After opening the file, we write zeros to the file by calling `fallocate` which also reserves the memory for the entire log. If `fallocate` is not available (e.g., for ext2), we use

`ftruncate`, which is noticeably slower. It is mandatory to reserve the memory for the entire log when creating/opening the file as write accesses that require enlarging the file and appending the data might use the page cache again.

The most important difference when accessing files directly is that every read and write access must be page- and block-aligned (typically, the page size is a multiple of the block-size). This means, both the position in file and in buffer must be page-aligned, as well as the end of the read/write access. In Sections V-E1 to V-F, we described the different disk accesses. In this section, we discuss the impact on the buffer and file position and how to handle the accesses correctly.

There are only two read access patterns: (1) read an entire segment from a secondary log and (2) read an entire version log. In both cases, both the file position (either 0 or a multiple of the segment size which is a multiple of the page size) as well as the buffer position (always 0) is page-aligned. Thus, for read accesses we do not have to consider the alignment. The function `pread` might return before all data was read. Therefore, all read accesses are executed in a loop which breaks if all data have been read or the end of the file has been reached.

We discuss the write access patterns separately:

Compacting a version log: Prior to the reorganization of a secondary log, the version log is read-in, compacted and written back to disk. In this case, the buffer position is page-aligned because the entire buffer is written (from buffer position 0). The file position is page-aligned as well because we write to file position 0 (see Figure 5 situation 1). If the end is not page-aligned (same for buffer and file), we write the entire last page (see Figure 5 situation 2). This requires the buffer being larger than the data. We discuss the buffer allocation which considers writing over the data boundaries at the end of this section. After the write access, the version log is truncated with `ftruncate`.

Writing to a new segment in secondary log: When writing to a new segment in a secondary log, the file position is page-aligned as the position is a multiple of the segment size (which is a multiple of the page size). If the buffer position is 0, the buffer can be written to the corresponding secondary log like the version log (see Figure 5 situation 1 and 2). Secondary logs are accessed segment-wise and within a segment, data is always appended. The end of a segment is also the end of a page. Therefore, writing to a segment does not affect the following segment, even if the last page is filled with invalid data to page-align the write access. The position in buffer is not always 0. Sometimes, a segment is filled up and the rest of the buffer is written to a new segment. In this situation, if the position is not a multiple of the page size, we move the complete data to the beginning of the page of the first byte (with `memmove`) and write to the file from this position in the buffer (see Figure 5 situation 3). The end is handled as before.

Flushing a version buffer to its version log: Whenever the version buffer is full or the threshold is reached, the entire version buffer is flushed to the end of the version log. In this case, the buffer is aligned, but the file position most likely is not as we append at the end of the version log and a version number has a size of 13 bytes. Therefore, when writing to the version log, all bytes from the last written page of the file have to be read and put in front of the data in the ByteBuffer to write (see Figure 5 situation 4). Then, the data is moved to the offset in file (start position $\%$ page size) within the ByteBuffer. All buffers have one additional free page in front of the start position for this situation (see the end of this section). Again, if the end position is not aligned, the last page must be written entirely and the file is truncated afterwards.

Writing to primary log: This is mostly the same situation as flushing to a version log. The buffer position is 0 and the file position arbitrary (see Figure 5 situation 4). However, the file is not truncated afterwards.

Appending to a segment in secondary log: When appending to a segment, both the position in buffer as the position in file is most likely not page-aligned. This is a combination of situation 3 and 4 in Figure 5. However, as all bytes to write in the buffer have to be moved anyway (in situation 4), the write access is handled like described in situation 4 of Figure 5.

Freeing a segment in secondary log: If all log entries of a segment are invalid during the reorganization, the segment must be marked as free. This is done by writing a 0 to the beginning of the segment. As it is not possible to write a single byte, we write an entire page filled with zeros.

All write accesses are executed in a loop because `pwrite` might return before all bytes have been written.

Buffer Allocation: The write accesses, as described, do not need allocations or to copy data to other buffers. In some cases, data is moved within the write buffer and data is read from file to the buffer beyond the boundaries of the buffer. This must be considered for the buffer allocation as well as the page alignment of the buffer. All buffers used for writing to disk are allocated in a wrapper class which stores the buffer's address and the ByteBuffer's reference. The ByteBuffers are allocated with `ByteBuffer.allocateDirect()` and the byte order is set to little endian. A ByteBuffer created with aforementioned method in most cases is not page-aligned. Hence, we create a ByteBuffer which is exactly one page larger than required. Then, we set the position to $address \% page\ size$ and the limit to $position + requested\ length$. Finally, we slice the ByteBuffer to create a second ByteBuffer instance which refers to the same byte array in native memory but with the position and limit of the first instance as beginning and capacity.

The ByteBuffers must not only be page-aligned, but also have one free page in front of it and the last page must be allocated entirely, as well, if the end is not page-aligned. Therefore, we add another page and the overlapping bytes to the size of the ByteBuffer. Furthermore, the address and the beginning of the sliced buffer is set to the page-aligned offset plus one page. The additional memory is not a problem because the buffers are rather large. For all buffers used in the logging module, we need less than 160 KB additional memory (see Section VI).

3) Raw Access: The RAW access is based on O_DIRECT and shares the read and write functions. The difference is that the direct access method uses files provided by the file system whereas the RAW access method accesses a raw partition instead. This way, we can reduce the overhead of a file system

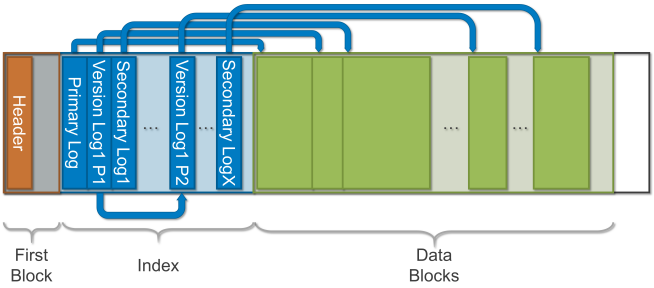

Figure 6: *Structure of the RAW partition*

like timestamps, many indirections and journaling. Furthermore, we can optimize the structure for the only purpose of storing logs (appends, no deletes).

The raw access requires structuring the partition to access different logs. We divide the raw partition into three parts (see Figure 6): a start block with a partition header (for identifying a partition after failure), an index for finding logs and a data block which stores the logs.

The index has a tabular form. Every row is a 64-byte index entry containing an address within the partition pointing to a log, the log's name, type (primary, secondary or version log) and size. One difficulty is that the version logs can grow and shrink (the other logs have a fix size). Therefore, an index can point to another index entry for indexing the next part of a log. Initially, a version log is created by appending one index entry and one 16 MB data block. If the version log grows beyond 16 MB, another index entry and 16 MB data block are created. To find the second part of the version log, the first index entry stores the address of the second index entry.

The index block is cached in RAM for fast indexing. When the index was changed, the manipulated entries are written to disk (page-wise). Obviously, write accesses to the index block must be synchronized. The read and write accesses to the logs must not.

4) Comparison of the Access Methods: The default access method in DXRAM is direct access. Compared to the RAW access, it is more versatile as logs can be stored on every partition with a file system. RAW requires a dedicated raw partition which cannot not be provided on every server. Furthermore, the written logs can be analyzed with other tools when stored as a file. Table I shows the dis-/advantages of using O_DIRECT in comparison with using the page cache. Generally, the disadvantages outweigh the advantages. However, DXRAM's demands are uncommon. First of all, write and read accesses are quite large. Reads are usually at least 8 MB, writes are as large as possible without impairing the reliability. During a typical load phase, write accesses are between 7 and 8 MB on average. Furthermore, a exemplary backup server stores seven times the amount it stores in RAM and most of the RAM is occupied by in-memory objects which strictly limits the size of the page cache. Thus, caching is not very effective for DXRAM's logging. On the contrary, the disadvantages of utilizing the page cache weigh much more. The double buffering is not efficient and the page cache cannot be restricted in size and may grow rapidly. Whenever the application needs more memory and all memory is in use, the page cache must be flushed to disk, which can take a while. Furthermore, if the page cache contains many dirty

TABLE I: DIS-/ADVANTAGES OF USING O_DIRECT

Advantages:
Lower and predictable RAM consumption (no caching)
Synchronous access without copying
Disadvantages:
More complex to use
No performance benefits from caching
Dependent on the underlying system
No asynchronous write access supported

pages, flushing the cache can pause the entire system for several seconds (amount of dirty pages can be configured). Another problem is the reliability. When a server crashes all dirty pages are lost. Since the DXRAM does not know when the write access is flushed to disk, DXRAM and its applications might be in an inconsistent state when rebooting. The RandomAccessFile allows synchronous disk access, but this is slow in comparison to the implemented access via O_DIRECT as all data has to be copied to the page cache anyway.

VI. LOGGING IN DXRAM - METADATA OVERHEAD

Table II shows the memory usage of all data structure used in DXRAM's backup system (logging, reorganization and recovery). When a backup server stores 1000 backup zones with an average chunk size of 64 bytes, resulting in around 1 TB on disk, the RAM usage would be 3.4 GB, which is around $\frac{1}{300}$ of the disk usage. All given values are optimized for performance, i.e., if the load is very high and every update belongs to the same backup zone, the performance would be optimal. The best way to reduce the memory usage is to shrink the version buffers. With 1 MB version buffers, DXRAM needs 1.4 GB or $< \frac{1}{700}$ of the disk space and the performance would be untouched for most situations (assuming that the access distribution is not extreme).

VII. RELATED WORK ON SEGMENT SELECTION

In [11], Rosenblum et al. presented a file system which is based on a log structure, i.e., file updates are appended to a log instead of updating in-place. This allows aggregating of write accesses in RAM in order to efficiently utilize the disk by writing large batches. For given workloads the log-structured file system (LFS) utilizes the disk an order of magnitude more efficiently than an update-in-place file system. The work was inspired by write-ahead logs of databases and generational garbage collection of type-safe languages which also need to clean-up in order to reclaim space by removing invalid/outdated objects.

While not being the first developing a LFS, Rosenblum et al. contributed by analyzing workloads to find an efficient reorganization scheme. A fast reorganization is important to keep a constant write throughput (provide enough free space for writes) and to allow a fast crash recovery (less invalid/outdated objects to process). In [11], a log is subdivided into 8 MB segments. The reorganization selects a segment, reorganizes it and proceeds with another segment. Important for the efficiency of the reorganization is the segment selection.

TABLE II: MEMORY CONSUMPTION FOR N BACKUP ZONES

Data Structure	Quantity	Aggregated Memory Consumption
Write Buffer	1	32 MB
Secondary Log Buffers	N	N∗128 KB
Version Buffers	N	N∗3 MB
Pooled Buffers for Secondary Logs	$8 * 8$ MB+$32 * 1$ MB+$64 * 0.5$ MB	128 MB
Pooled Read/Write Buffers for Version Logs	2	$\sim$ 50 MB
Pooled Buffer for Reorganization	1	8 MB
Pooled Buffers for Recovery	5	40 MB
Range Sizes Hash Table	1	28 KB
Range Buffers Hash Table	1	44 KB
Version Hash Table for Reorganization	1	$\sim$ 45 MB

Optimally, the segment with most invalid/outdated data is selected for the cleaning. Rosenblum et al. stated the assumption that "the older the data in a segment the longer it is likely to remain unchanged" [11]. This leads to the following cost benefit formula, which did well in the evaluation:

$$\frac{benefit}{cost} = \frac{free\ space\ generated * age\ of\ data}{cost} = \frac{(1-u) * age}{1+u} \quad (1)$$

u is the utilization of the segment which is the fraction of data still live, age is the age of the youngest block within a segment.

Seltzer et al. did a more thorough performance analysis on log-structured file systems showing the high performance impact of the cleaning (more than 34 % degradation if cleaning is necessary) [12]. In [7], Ousterhout et al. applied the ideas of a LFS for the in-memory key-value store RAMCloud. RAMCloud uses a log for storing in-memory objects and replicates the objects segment-wise to remote disks. Furthermore, they present a two-level cleaning approach which is a combination of in-memory reorganization and disk compactification. In RAMCloud, all complexity resides on the masters, storing the objects in RAM. The backup servers are used for the plain writing to disk. Therefore, the backup servers cannot execute the reorganization (they miss information like the current version numbers), but they can compact logs from time to time. To avoid recovering already deleted objects, RAMCloud's masters write *tombstones* (difficult to remove) to the logs. DXRAM, on the other hand, stores the in-memory objects with a tailored memory management in RAM and replicates the objects to backup servers as soon as they are written. The backup servers perform the version control and reorganization of all of its stored objects without communicating with masters. DXRAM avoids tombstones as backup servers can identify deleted objects through the version control.

In [13], Rumble et al. modified the cost benefit formula for RAMCloud:

$$\frac{benefit}{cost} = \frac{(1-u) * segmentAge}{u} \quad (2)$$

The first difference, regarding the denominator (from $1+u$ to u), considers that RAMCloud does not have to read the segment from disk prior to the reorganization as all segments are stored in RAM on masters. The second change concerns the age which is not the age of the youngest block, anymore, but the average age of all objects in a segment. The latter avoids the unnecessary reorganization of segments which have a high utilization and store mostly old objects but one or a few new objects.

VIII. SEGMENT SELECTION IN DXRAM

In DXRAM, the segment selection requires two steps because DXRAM does not store one log with all objects of a master but one secondary log per backup zone (a master can have hundreds of backup zones). The first step is selecting a secondary log to be reorganized and the second step is selecting segments of this secondary log. The secondary log to reorganize is chosen by its size: the largest log has the most invalid data as backup zones are identical in size (assuming there is no fragmentation). For selecting a segment, we cannot adopt RAMCloud's approach for DXRAM because instead of an in-memory log on the masters DXRAM uses an in-place memory management on the masters and a separated log-structure on backups. Therefore, the log entries on the backups are never read individually but as whole segments. This allows us to spare storing the locations of log entries within a log saving a lot of memory on masteres (e.g., for one billion 64-byte chunks and three replicas: $>$ 30 GB per master are saved). But, without the location of invalid log entries, it is not possible to determine the fraction of live data of a segment. Obviously, searching for invalid versions to update the segments' utilizations is not an option.

In the following we use a different definition of the term *utilization* (in comparison to [11] and [13]). We define the utilization u as the plain filling degree of a segment (live, outdated and deleted chunks).

1) Basic Approach: A secondary log is never reorganized as a whole but incrementally by reorganizing single segments (default: 8 MB). Similar to the secondary log selection, the segment selection tries to find the segment with the most outdated data. In the basic approach, we calculate a segment's age based on its creation and last reorganization and select an old segment with high utilization for cleaning ($max(age * utilization)$). We think this is a good metric as there is a

higher probability of finding outdated objects in segments that are large and have not been reorganized for a longer period of time. Additionally, this approach is very simple to implement and comes at no cost as all required metadata is already available.

2) Advanced Approach: The advanced approach tries to improve the log selection, i.e., selected segments contain the most outdated data, by including additional or more precise indicators. The decision making must consider the following constraints: (1) neither the exact location of an object nor the segment an object is stored in is available because (2) object specific information cannot be stored in RAM due to the memory consumption being too high. Therefore, (3) the utilization as described in [11] and [13] representing the fraction of valid data is not available, either, because maintaining the information would require the location of previous versions.

Utilization: The utilization (filling degree) is a good indicator for the segment selection if all chunks are accessed evenly because the segment with highest utilization would, on average, have the most invalid data. But, segments with much cold, long-living data are chosen repeatedly blocking the reorganization of segments with (more) invalid data. Therefore, the utilization alone is not a good indicator in every scenario.

Age: In Section VIII-1, we defined the age of a segment as the time since the last reorganization or creation. While this approach is easy to implement, the validity of the age is highly limited as the age of long-living objects is not covered (the time is reset regardless of whether much data was discarded or not) and a freshly reorganized segment is not necessarily used next to add new objects. The least recently reorganized segment might even store the same still valid objects. In this section, we discuss an approach to determine a segment's age based on the age of all containing objects.

Rosenblum et al. state that "the older the data in a segment the longer it is likely to remain unchanged" [11]. This claim cannot be transferred to DXRAM because it is based on the assumption that updated and deleted data is marked invalid in the segment headers and the age is determined for the valid data, only. In DXRAM, instead, invalid data is exclusively detected and discarded during the reorganization, i.e., a segment's age is the average age of all, valid and invalid, objects. Without the implication regarding the validity of an object, an object's age has to be interpreted differently: typically, older objects are more likely to be deleted or updated. Thus, a segment with more old objects might be the better choice for the reorganization. But, often objects can be split into the two categories: hot and cold data. Cold data consists of long-living objects that are unlikely to be replaced/removed. Therefore, a segment with very old objects might not be the best choice. Altogether, the age of an object is an important indicator for the validity of an object.

Average age per entry: To get a more accurate representation of a segment's age, we store a 4-byte timestamp in the log entry header of all objects (stored in front of the object on disk). An empty segment has the age 0. After the reorganization of a segment its age is defined by:

$$a_{seg_i} = \frac{\sum_{j=0}^{n} t - t_{c_j} | c_j \ valid}{m} \tag{3}$$

n is the number of objects in segment i and m the number of valid objects (c for chunk). As every object ages between two reorganizations, when selecting a segment, we adjust the average age of a segment by adding the time since its last reorganization.

$$a'_{seg_i} = a_{seg_i} + (t - t_{reg}) \tag{4}$$

Assuming we add a new object (c_x) at the same time the reorganization is executed, then the age is modified in the following way:

$$\begin{aligned} a_{seg_i} &= a_{seg_i} + \frac{t - t_{c_x} - a_{seg_i}}{n+1} \\ &= a_{seg_i} - \frac{a_{seg_i}}{n+1} \end{aligned} \tag{5}$$

When adding an object after the reorganization, we have to consider the time since the last reorganization to avoid increasing the age too much during the segment selection as a segment's age is based on all objects' ages at the time of the last reorganization. The object did not exist at this time. Therefore, we have to subtract the time difference.

$$a_{seg_i} = a_{seg_i} - \frac{a_{seg_i} + (t - t_{reg})}{n+1} \tag{6}$$

Average age per byte: Objects might differ significantly in size. To avoid missing segments with large old and invalid objects (to be discarded) and many small young objects (decreasing the age), we calculate the age per byte and not per chunk. Furthermore, we exclude every object which is older than a predefined threshold and still valid (hot-to-cold transformation).

$$a_{seg_i} = \frac{\sum_{j=0}^{n} (t - t_{c_j}) * s_{c_j} | c_j \ valid \ and \ t - t_{c_j} < t_{max}}{\sum_{j=0}^{m} s_{c_j}} \tag{7}$$

$$a_{seg_i} = a_{seg_i} - \frac{(a_{seg_i} + (t - t_{reg})) * s_{c_x}}{u_{seg_i} * s + s_{c_x}} \tag{8}$$

s is the segment size (e.g., 8 MB) and s_{c_j} the size of $chunk_j$. u_{seg_i} is the utilization (filling degree) of the segment.

Utilization & Age: The final segment selection is based equally on the utilization and age of a segment:

$$seg = i \in \{1, ..., l\} | max(u_{seg_1} * a'_{seg_1}, ..., u_{seg_l} * a'_{seg_l}) \tag{9}$$

l is the number of segments of the secondary log, excluding segments which have not been used yet (at the end of the log).

Timestamps: The used 4-byte timestamps show the elapsed seconds since the secondary log creation. An overflow occurs after more than 68 years and affects the segment selection (wrong decisions) for a short time, only.

IX. Related Work on Copysets

In [14], Cidon et al. present a replication scheme which, in comparison to random replication, significantly reduces the frequency of data loss events in exchange for a larger amount of lost data in case of a data loss event. The authors motivates that restoring the data after a data loss event has fixed costs regardless of the amount of lost data. Thus, losing a large amount of data seldom is more attractive for cluster operators than losing small amounts frequently. The probability for data loss is rather high with random replication in large clusters because, assuming the number of objects is high, every master most-likely stores replicas on every available backup server. Thus, a failure of x backup servers, where x is the number of replicas for every object, results in a data loss event. The basic idea in [14], is to limit the number of backup servers one master replicates its data to. The limited set of available backup servers for a master is called a `copyset`. Subsequently, data loss is possible, if a set of x backup servers of one copyset crash, only. Assuming the number of backup servers per copyset R (which is also the replication factor) is much lower than the total number of backup servers N, the probability will be much lower. The authors exemplify two scenarios to prove their statement: (1) in a 5000-node RAMCloud cluster, copyset replication reduces the data loss probability from 99.99% to 0.15%. (2) In a HDFS cluster with a workload from Facebook, the probability is reduced from 22.8% to 0.78%.

The copysets are created by permuting the N backup servers and assigning R consecutive backup servers from the permutations to a copyset. The number of permutations P is determined by:

$$P = ceil(S/(R-1)) \tag{10}$$

R is the number of servers per copyset and S is the scatter width that defines the number of backup servers one master replicates its data to. If the scatter width is higher, more permutations are generated which results in one server being in more copysets.

Example: $N = 6, R = 3, S = 4$. The number of permutations is 2 then and two permutations could be $5, 1, 3, 4, 6, 2$ and $6, 1, 2, 3, 4, 5$. Hence, the copysets are $\{5, 1, 3\}$, $\{4, 6, 2\}$, $\{6, 1, 2\}$ and $\{3, 4, 5\}$. To determine the backup servers for an object, the first backup server is chosen randomly. Afterwards, one copyset that includes the randomly chosen backup server is selected and the other servers in the copyset are assigned as additional backup servers. In the example above, if backup server 3 is chosen randomly, the other backup servers are either 5 and 1 or 4 and 5 (scatter width in example is 3, only, but with $N >> S$ a scatter width of 4 is very likely). Every primary backups' files are distributed to the same set of backups (1, 4 and 5). Only, if all nodes from one copyset fail simultaneously data loss occurs. The scatter width is important for the recovery as it defines the number of servers which can recover a failed server in parallel.

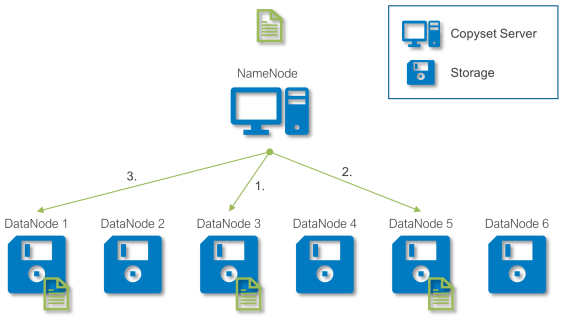

Figure 7: *Copyset Determination in HDFS*

In the next two sections, based on X, we further discuss copysets on two systems: HDFS and RAMCloud [14]. In Section X-B, we use the same example to present the copysets implementation of DXRAM.

A. Copysets in HDFS

The Apache Hadoop Distributed File System (HDFS) is based on the Google File System (GFS) [15] but has significant differences regarding the node allocation and chunk location management. HDFS is part of the open source programming MapReduce framework Hadoop. HDFS was designed to run on commodity hardware and applications with big data sets [16]. It has a master-slave architecture with a single master, called NameNode, and multiple slaves. The NameNode is responsible for the metadata and the slaves, also called DataNodes, for storing the data. For fault-tolerance reasons the data is replicated to other DataNodes, in 64 MB blocks.

When using copysets for HDFS, the NameNode creates the copysets at system startup like described in the previous section. Every time a new file has to be stored, the NameNode chooses the first location randomly and then $R-1$ DataNodes belonging to one of the copysets the first DataNode is in (Figure 7).

When a new DataNode is added to the system, the NameNode generates $S/(R-1)$ new copysets which contain the new server. When a server crashes, it is replaced randomly in all copysets. In the example from the previous section, if DataNode 2 crashes, the copyset could be modified in the following way: $\{5, 1, 3\}$ $\{4, 6, ~~2~~ 3\}$ $\{6, 1, ~~2~~ 5\}$ $\{3, 4, 5\}$.

Subsequently, if another DataNode with ID 7 is added, two additional copysets would be generated, for example: $\{7, 1, 4\}$ and $\{3, 7, 5\}$.

B. Copysets in RAMCloud

RAMCloud is described in Sections III and VII.

In RAMCloud copysets are created on the coordinator. Whenever a master creates a new in-memory object, it queries a set of backup servers from the coordinator (Figure 8). The first backup server is chosen randomly, the others belong to a copyset containing the first backup server. Every primary backups' objects are distributed to the same set of backup servers. But, every masters' in-memory chunks are scattered across

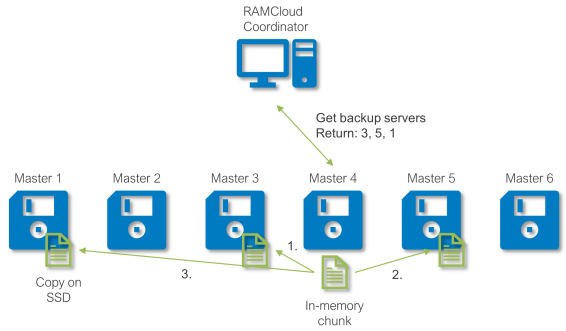

Figure 8: *Copyset Determination in RAMCloud*

the cluster. Only, if all nodes from one copyset and the master fail simultaneously data loss occurs. The scatter width is **not** important for recovery as every master replicates its chunks to many copysets. Therefore, it is always $S = R - 1 = 2$ and the recovery time is nearly unaffected (1.1s instead of 0.73s [14]).

In HDFS for every new DataNode $S/(R-1)$ copysets are added. In RAMCloud, one has $S/(R-1) = 1$ and instead of creating one new copyset directly, a new copyset is created when three new servers joined (all three servers are in the new copyset). When a server fails, it is replaced randomly like in HDFS.

X. Copyset Replication in DXRAM

In this section, we describe the most relevant aspects of DXRAM's backup zones in Section X-A, followed by the copyset implementation of DXRAM in Section X-B.

A. Backup Zones

In order to enable a fast parallel recovery, in DXRAM, the chunks of one server are partitioned into several backup zones (with a size of 256 MB) which are scattered across potentially many backup servers (e.g., a 64 GB server assigned with 256 different backup servers). Every server determines its own backup zones and informs its associated superpeer on each backup zone creation. This approach avoids global coordination regarding backup zone selection between servers. We use a replication factor of three by default but it is configurable.

Each backup zone is identified by a zone ID (ZID). The ZID alone is not globally unique but it is in combination with the creator's node ID derived from the context. A new backup zone is created whenever a chunk does not fit into any existing backup zone. If chunks were deleted, a backup zone will be gradually refilled with new chunks. Furthermore, chunks with reused CIDs are stored in the same backup zone as before, if possible, to minimize meta-data overhead. Three backup servers are assigned to each backup zone with a fixed replication ordering guaranteeing consistency. According to the ordering, the first backup server receives all backup requests first, the second afterwards and so on. Furthermore, backup requests are bundled whenever possible. If there are less than three servers currently available for backup (e.g., during startup), the next joining server will be used and receives all previously replicated chunks of this zone.

A server notifies its superpeer whenever a new backup zone was created or a backup server was changed. This results in a single message for every 256 MB (e.g., once after 3.5×10^6 64-byte chunks have been created) and a few messages per server failure (the failed backup server has to be replaced), only. To further reduce memory consumption on superpeers (resulting in just 10 bytes per backup zone in the best case), a superpeer does not store backup zone affiliations of chunks. This information is exclusively stored on the owner of a chunk as only this server must know the corresponding backup zone of its chunks for sending backup updates.

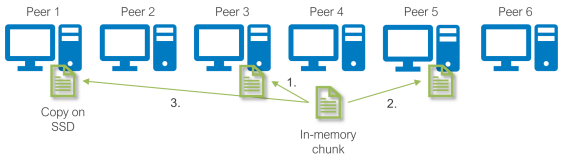

Figure 9: *Copyset Determination in DXRAM*

B. Copysets in DXRAM

DXRAM does not have a coordinator, like the NameNode in HDFS or the coordinator in RAMCloud, but a set of servers responsible for the metadata (superpeers) and another set responsible for storing the data and backups (peers). Hence, we decided to create the copysets on every master independently but consistently by using the same input and algorithm to create copysets (no coordination needed). Consequently, every master also determines its own backup servers accordingly by choosing the primary backup server randomly and all other backup servers from one copyset containing the primary backup server (Figure 9). Optionally, the primary backup server can be selected disjunctive and/or locality-aware. Another important difference is that DXRAM determines backup servers not for single chunks but for backup ranges containing many chunks (e.g., 256 MB). Therefore, the maximal number of copysets is smaller, if random replication is used. Still, with copyset replication the probability for data loss can be reduced.

For joining servers, we use the same strategy as RAMCloud: we wait for R new servers to join and, then, create a new copyset containing all three servers. When a server crashed, it is replaced in all copysets. However, because of the decentralized copyset determination, we have to replace the failed server consistently on all masters. We do this, by using a seed which is based on the copyset (aggregated node IDs) for the pseudo random number generator.

The initial copyset determination is based on the nodes-file (a file used for startup which lists all servers participating) which is identical for all servers. Further un-/available servers are propagated by join and failure events which are distributed among superpeers first and to the peers afterwards. But, copysets can differ when servers are added because masters might detect the joining servers in different order. This case is rather unlikely but can occur from time to time. Therefore, the number of copysets (globally) can be higher than N/R but still is a lot smaller (for $N >> S$) compared to random replication which is $\binom{N}{R}$ (e.g., with 512 backup servers $>=$ 171 for copyset replication and $\binom{512}{3} = 22,238,720$ for random replication; the number of combinations is limited by the number of backup zones in the system, for example 262,144).

In DXRAM, copyset replication can be combined with additional replication schemes like disjunctive first backup servers (to increase the throughput of the parallel recovery) and/or locality-awareness.

XI. Evaluation

In this section, we evaluate the byte array access methods as well as the disk access methods. Furthermore, we provide a thorough performance analysis on the logging and reorganization of DXRAM. The latter also includes a comparison of both presented segment selection strategies.

All tests were executed on our non-virtualized cluster with 56 Gbit/s InfiniBand connection and servers with PCI-E nvme SSDs (400 GB Intel DC P3600 Series), 64 GB RAM, Intel Xeon E5-1650 CPU (six cores) and Ubuntu 16.04 with kernel 4.4.0-64.

A. Byte Array Access

Log entries are almost always aggregated in larger buffers in the logging module. In order to find the best way to handle these buffers, we evaluated the different byte array access techniques provided by Java. We wrote a benchmark which writes to and reads from 8 MB buffers by using the access specific methods. The techniques are:

- DirectByteBuffer BE: A ByteBuffer allocated outside the Java heap with big endianness.
- DirectByteBuffer LE: A ByteBuffer allocated outside the Java heap with little endianness (native order).
- HeapByteBuffer BE: A ByteBuffer allocated in the Java heap with big endianness (order of Java heap).
- HeapByteBuffer LE: A ByteBuffer allocated in the Java heap with little endianness.
- Array: A byte array in Java heap.
- Unsafe: A ByteBuffer allocated outside the Java heap with little endianness, accessed with methods provided by sun.misc.Unsafe.

Every buffer is filled first and then read entirely. We write/read a long value, followed by a short and three byte values, which is the access pattern of the version buffer and is also very similar to the access patterns of the primary and secondary log buffers. For representative results, we fill and read the buffers 1,000 times and ignore the first 100 iterations. In every iteration, we access another buffer. The buffers are allocated at the beginning of the benchmark to simulate the buffer pooling. For Java's Unsafe access, we also do boundary checks before every read and write access. Every test was executed five times.

Figure 10 shows the results of the presented benchmark. The benchmark runs are very consistent for all access methods but the DirectByteBuffer with big endianness. The native memory order on the used server is little endian. Therefore, the high variance can be explained by the byte swapping prior to every write access which is a rather CPU intense step. On the contrary, the Java heap is big endian. Thus, the variance of the HeapByteBuffer with little endianness is also higher than

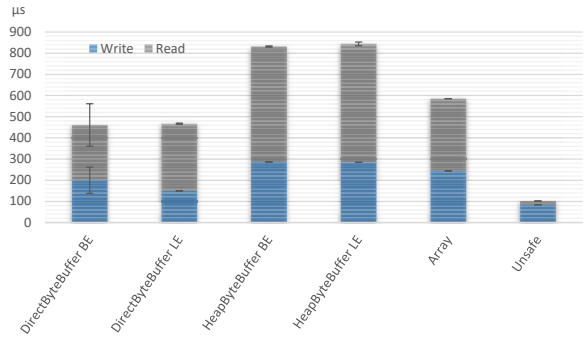

Figure 10: *Evaluation of different byte array access methods. Writing and reading an 8 MB buffer (one to eight bytes per access) 900 times*

with big endianness. But, the difference is minor in this case. Nonetheless, using the endianness of the underlying memory for the ByteBuffer is advisable.

The DirectByteBuffer performs considerably better than the HeapByteBuffer and the heap array. Manipulating the data with Unsafe is even faster than with the DirectByteBuffer's methods. Subsequently, for the RandomAccessFile which needs a heap array for writing and reading, the fastest technique is the array itself. For O_DIRECT and RAW access which requires the data to be off Java heap, Unsafe is the fastest choice. However, as the performance is relatively close, e.g., writing an entire 8 MB segment with longs, shorts and bytes is 120 ns slower with a DirectByteBuffer than Unsafe, and in order to reduce complexity (no wrapper, branching, dedicated serialization methods or boundary checks) and increase maintainability (debugging of segmentation faults is bothersome, future of Unsafe is unclear), we use ByteBuffers (DirectByteBuffer LE for O_DIRECT/RAW and HeapByteBuffer BE for RandomAccessFile) for the logging module of DXRAM. Furthermore, one has to consider that this are the results of a micro benchmark and the real application's behavior is not identical.

B. Logging and Reorganization

In this section, we evaluate the logging and reorganization performance of DXRAM. First, we analyzed the maximum throughput of the SSD first. We used a SSD of the type Intel DC P3600 Series with a capacity of 400 GB. It provides a maximum throughput of 2.6 GB/s for read accesses and 1.7 GB/s for write accesses. The random I/O throughput of 4 KB chunks is capped at 450 MB/s for reads, 56 MB/s for writes and 160 MB/s for 70% reads and 30% writes. We measured the SSD performance with `dd` by writing 1,024 8 MB (default segment size) files (`/dev/zero`) with direct access.The results for 8 MB write accesses are significantly below the maximum throughput, showing 914 MB/s. With two processes writing concurrently, the throughput improves to 1,116 MB/s. With more processes the throughput is consistent (e.g., 1,170 MB/s with four processes). SSDs operate highly parallel and both the nvme driver (no I/O scheduler) and the Linux kernel (Multi-Queue Block IO Queueing [17]) take advantage of that if read/write accesses are executed in parallel. DXRAM benefits

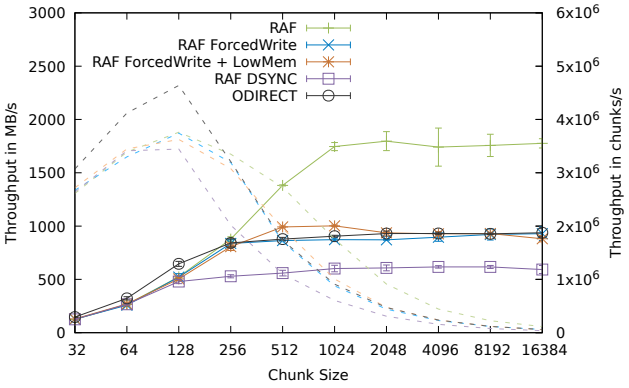

Figure 11: *Evaluation of different disk access methods. Every chunk is written once, in sequential order. Solid lines: throughput in MB/s, dashed lines: throughput in chunks/s*

from the parallelism by logging, reorganizing, recovering and reading/writing versions concurrently. When writing smaller chunks to disk, the throughput degrades, e.g., 477 MB/s for 8 KB chunks.

1) Logging: Figure 11 shows the logging throughput (in MB/s and chunks/s) of the RandomAccessFile and O_DIRECT for chunk sizes from 32 bytes to 16 KB. We evaluated the RandomAccessFile in four different configurations explained in the next paragraphs: (1) without limitations (RAF in Figure 11), (2) with forced writes (RAF ForcedWrite), (3) with forced writes and with limited memory available (RAF ForcedWrite + LowMem) and (4) in synchronous mode (RAF DSYNC).

In order to provide resilience, DXRAM requires to store logged chunks persistently without much delay. However, In the default configuration, the OS caches many dirty pages in the page cache before eventually flushing the pages to disk. Therefore, when evaluating with forced writes, the OS is configured to flush dirty pages of the page cache reaching 8 MB, if possible, and immediately flush when 32 MB is dirty. The OS's flushing threads are also configured to flush more frequently (every 100 ms) than normal regardless of the two thresholds.

Even with a limited amount of dirty pages in the page cache, the page cache might grow critically for a memory-heavy application, i.e., the page cache occupies memory the application needs requiring to flush the page cache which might require many seconds. Therefore, we tested the logging with a limited amount of memory available by starting a program apriori which occupies most of the memory (92.5%/59.2 GB).

Finally, the RandomAccessFile was opened in synchronous mode (`rwd`). In this mode, a write access returns after the data was written to disk. In contrary to `rws`, the file system's metadata (e.g., timestamps) might not have been updated. In all other test, the RandomAccessFile was opened with `rw` which is asynchronous.

The benchmark used to determine the logging throughput (and reorganization throughput in Section XI-B2), creates ten chunks (number configurable) and serializes them into a DirectByteBuffer. The buffer is passed to the logging component to be logged. For the next iteration the chunk IDs are incremented and the buffer is logged, again. This is repeated until the predefined number of chunks (e.g., 400,000,000 32-byte chunks) has been logged. The benchmark does not involve the network or any other components or services from DXRAM but the logging component. Every experiment is executed three times and old logs are removed and the SSD is trimmed (`fstrim`) between runs to get consistent results.

The RandomAccessFile without limitations (RAF) is the fastest disk access mode for chunks larger than 256 bytes. The disk is saturated at 1.7 to 1.8 GByte/s with 2 KB (and larger) chunks. The throughput is even higher in some cases than the maximum throughput specified by the manufacturer showing that not all data has been written to disk when the benchmark was finished. Additionally, the good performance comes at the cost of the page cache using more than 30 GB of the main memory. When limiting the amount of dirty pages, as expected, the performance degrades to around 1 GB/s for large chunks. Increasing the memory pressure does not further degrade the performance in this scenario because the logged data is never read, rendering the read cache useless (it is still larger than 30 GB). Using the RandomAccessFile in synchronous mode has a large penalty on the throughput, which is reduced to around 600 MB/s. When writing to disk with O_DIRECT, the access is synchronous as well, but the performance is considerable better than the synchronous RandomAccessFile as double buffering is prevented. Actually, up to 256-byte chunks the logging throughput is better than all RandomAccessFile configurations mostly due to the DirectByteBuffer being faster than the HeapByteBuffer (see Section XI-A). For the targeted chunk sizes of 32 to 256 bytes, DXRAM is able to log more than three million chunks per second, peaking at around 4.64 million 128-byte chunks per second. With around 930 MB/s for 2 to 16 KB chunks the DXRAM's logging performance with O_DIRECT is equal to copying 8 MB chunks with dd and a single thread (914 MB/s) but is much faster than copying 8 KB chunks with dd (477 MB/s).

2) Reorganization: For evaluating the reorganization or more specific the logging performance when the reorganization runs concurrently, we use four different access distributions: sequential, random, zipf and hotNcold (see enumeration below). The reorganization tests have two phases. First, all chunks are written sequentially to disk (equal to the logging test). Second, twice as many chunks are updated according to the access distribution. For example, when using the sequential distribution, all chunks are written three times in sequential order. With random distribution, on the other hand, all x chunks are written sequentially and then $2 * x$ chunks are chosen randomly to be updated. Chunks are written in batches of ten to reduce the overhead for the benchmark itself. The batch size does not affect the logging performance because every chunk needs to be processed solely by the log component (to create a log entry header with unique version, checksum and more).

1) Sequential: Updating the chunks in ascending order from first to last in batches of ten. Repeat until number of updates is reached.
2) Random: Choosing a chunk randomly and update it with the nine following chunks (locality). If the randomly chosen chunk is at the end of a backup zone, the nine

preceding chunks are updated. Repeat until the number of updates is reached.

3) Zipf: Every chunk has an allocated probability to be selected according to the zipf distribution. Select nine succeeding or preceding chunks to complete the batch. Repeat until the number of updates is reached.
The zipf distribution follows Zipf's empirical law which is a power law probability distribution studied in physical and social sciences. The zipf distribution allocates the frequency (probability to be chosen) inversely proportional to the rank in the frequency table. The nth most likely chosen element has the probability:

$$\frac{1}{\sum_1^N \frac{1}{n^e}} * \frac{1}{n^e} \tag{11}$$

with e being the skew. The benchmark has a freely selectable skew, but we consistently used 1 (harmonic series) for the evaluation which is close to the distribution in social media networks [18]. With the skew 1, the first element has a probability of nearly 7% to be chosen, the second 3.5%, the third 1.7%.
Efficiently accessing chunks with zipf distribution requires to generate the distribution before starting phase 2 of the benchmark because calculating the distribution on-the-fly is either too slow or mitigates the distribution. In [19], the authors present a fast method to choose elements according to the zipf distribution without creating the distribution apriori. However, this method allocates the highest probability to the first element, the second highest to the second element and so on. Thus, the values need to be hashed for scrambling the elements. This might destroy the zipf distribution if the hash function does not scatter uniformly (the value range is user-defined). Instead, we create two arrays prior to phase 2 of the benchmark. The first array contains the aggregated frequencies for all chunks, i.e., the value at index x is the probability of choosing the elements 0 to $x-1$ according to the zipf distribution. The second array is a permutation of all chunk IDs. To choose a chunk, a random value p in $[0.0, 1.0)$ is generated. Afterwards, we search for p or the succeeding value within the first array (binary search). The index i of the searched value is used to index into the second array. Finally, the chunk ID at i is selected to be updated.

4) HotNcold: Divide all chunks into two partitions: hot and cold. The hot partition contains 10% of the chunks (the cold 90%) and 90% of all updates are chosen from the hot partition (10% from the cold partition). Select nine succeeding or preceding chunks to complete the batch. Repeat until the number of updates was reached.
We create two arrays prior to phase 2 of the benchmark. The first array has $N*0.9$ entries and the second $N*0.1$. We store all cold chunks in the first array and the hot chunks in the second array. Whether a chunk is hot or cold is decided by generating a random value in $[0.0, 1.0)$ for every single chunk. If the value is < 0.1 it is considered hot and its chunk ID is stored in the second array. Otherwise, the chunk is cold and stored in the first array. During phase 2, a chunk is chosen by generating a random value in $[0.0, 1.0)$. If the value is < 0.9, we choose a random chunk from the second array (hot), otherwise we choose a chunk from the first array (cold).

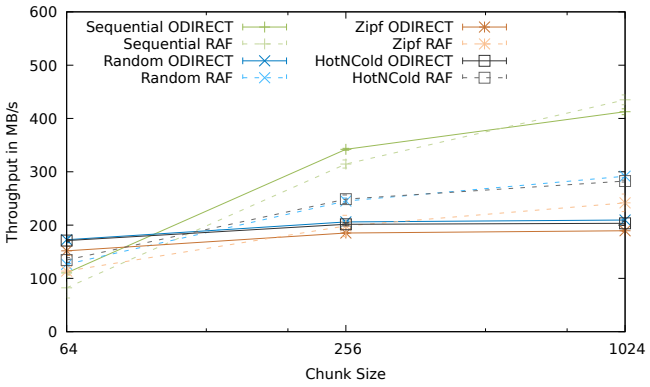

Figure 12: *Evaluation of the reorganization. Every chunk is written once, in sequential order. Then, twice as much updates are written according to given distribution*

Figure 12 shows the logging throughput during the reorganization test with 200,000,000 64-byte, 50,000,000 256-byte and 12,500,000 1024-byte chunks stored in 56 backup zones. As chunks are in average updated three times during all runs, the reorganization has to free space for updates to be written. More precisely, during phase 1 the reorganization idles as none of the secondary logs exceed their backup zone size, which indicates that the logs have no invalid data. In phase 2, the reorganization must free at least the amount of chunks written in phase 1 to have enough space in the logs to write all updates. We compare the RandomAccessFile with O_DIRECT. In contrary to the logging tests, we evaluated the RandomAccessFile with forced writes and memory pressure (between 87.5 and 92.5% depending on the memory consumption of the distribution), only, because the other configurations are not applicable or too slow for real-world applications. Phase 1 is not included in the throughput measurements.

Again, the direct access is considerably faster for small chunks. For 64-byte chunks, around $2 * 10^6$ chunks can be written to disk per second for all distributions. With larger chunks, the RandomAccessFile surpasses O_DIRECT for all distributions but the sequential distribution. This is because (1) the write accesses for each backup zone are much smaller because they are scattered across all backup zones. Therefore, buffering write accesses in the page cache improves the throughput (but increases the probability of data loss). (2) The page cache stores frequently accessed pages of the disk which can improve the throughput of the reorganization, too. But, this comes at a high price because the page cache puts the system under a high memory pressure which can lead to processes even being killed by the operating system what happened several times during the evaluation. From here on, all tests were executed with O_DIRECT access as the partly better performance of the RandomAccessFile is outweighed by the problems described before and due to DXRAM being design for very small chunks which are logged/reorganized faster with O_DIRECT.

For all distributions but the sequential distribution, the performance degradation in comparison to the logging test is caused by the arbitrary access to backup zones which make the aggregation much less efficient. For example, if 200,000,000 64-byte chunks are being accessed randomly, the

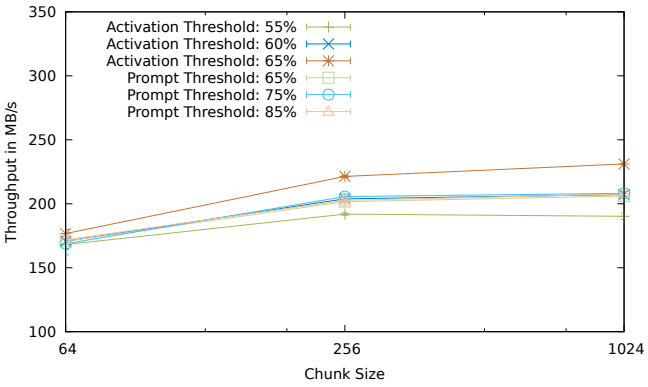

Figure 13: *Evaluation of the reorganization thresholds with random distribution.*

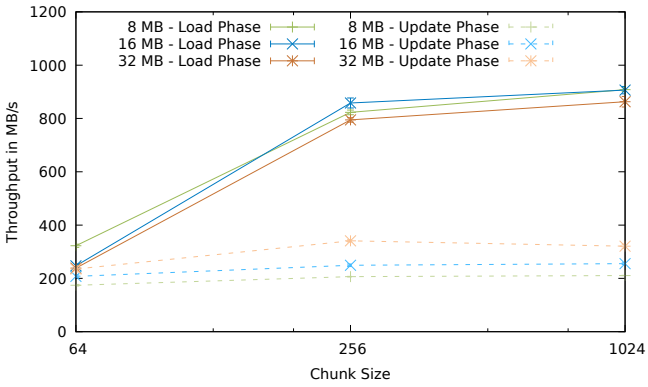

Figure 14: *Evaluation of the reorganization with different segment sizes*

probability is very high that all 56 backup zones are contained in every flush of the write buffer. During the loading phase (sequential), at most two backup zones are contained producing much larger write accesses. During all runs with random and hotNcold distributions, not one log was filled-up. With the zipf distribution the writer thread was blocked once in a while due to the logs storing the hot spots being full (the two to three most frequently updated chunks). Therefore, the zipf distribution is a little slower than the random and hotNcold distributions. For larger chunks, the three distributions are still restrained by the scattered access. The throughput of the sequential distribution, on the other hand, is dictated by the reorganization throughput. Thus, the throughput is worse for small chunks but improves significantly for larger chunks due to the reorganization being more efficient for larger chunks.

In order to log two million 64-byte chunks per second in phase 2, the reorganization has to free around two million chunks per second as well. With a utilization of 80% this results in reading 5.33 million and writing 4 million chunks per second. Additionally, version numbers have to be read from disk for the reorganization and written to disk after the reorganization and during the logging. For 64-byte chunks, this are around 3.3 million version numbers per log (without invalid entries).

Activation of the Reorganization: We implemented three mechanisms to activate the reorganization: (1) if a log is larger than a given threshold (e.g., 60%), it is available for the periodic reorganization, which selects the largest log for reorganization. (2) If the log size exceeds another threshold during writing to it (e.g., 80%), the writer thread prompts the reorganization by registering a reorganization request for the specific log. The reorganization thread prioritizes requests over the largest log, but finishes reorganizing the currently selected log first. At last, (3) if the writer thread is not able to write to a log because it is full or the fragmentation is too high to write all log entries, the writer registers an urgent reorganization request and awaits its execution. Urgent requests have the highest priority and are processed as soon as possible.

Figure 13 shows the logging throughput for a random distribution with varying activation (case 1) and prompt (case 2) thresholds. An activation threshold of 65% and an prompt threshold of 75% is the best choice in this scenario. With a higher activation threshold (beyond 65% was not tested), logs are reorganized too late which increases the pressure on the reorganization. If many logs breach the threshold in a short time, the reorganization cannot keep pace. With a lower activation threshold, too much work is done with a low utilization which is not efficient. With 55% an average of 2.98 MB are freed per reorganized segment, with 65% 3.95 MB. With a large prompt threshold, the reorganization might miss reorganizing a filling up log. If the prompt threshold is too low, the request queue might grow large and not necessarily the log with most pressure is reorganized but the first reaching the threshold (could still be 65% whereas another log could be at 95%, for instance). We uses 60% activation and 75% prompt thresholds throughout all other tests.

Segment Size: We also evaluated the impact of the segment size on the logging and reorganization performance (Figure 14). Interestingly, the reorganization benefits from larger segment sizes whereas the logging performance degrades. Larger segments allow the reorganization to process more log entries between I/O accesses which improves the performance. During the loading phase (sequential distribution), write accesses can be aggregated very efficiently because all chunks in the write buffer belong to one or two backup zones, only. However, while the average write access size with 8 MB segments is around 76% of the segment size (6.06 MB), it is reduced for 32 MB segments to 51% (17.23 MB; with 16 MB segments: 54%, 9.06 MB). This results in more often stocking up the larger segments which is slower than writing to the beginning of a segment as the data likely needs to be moved within the buffer. This also affects the reorganization, i.e., segments with higher utilization are beneficial for the reorganization. In all other tests, we use a segment size of 8 MB because it is a good compromise between logging and reorganization performance (especially for small chunks) and it has the lowest memory consumption as pooled buffers are smaller.

Two-Level Logging: In Figures 15 and 16, we evaluated the two-level logging by varying the secondary log buffer sizes. The size of the secondary log buffers impacts the logging significantly as it defines the threshold for log entry batches to be written to secondary log or to the primary log and secondary log buffer. For example, if the secondary log buffers have a size of 128 KB, all sorted and aggregated batches from the write buffer smaller than 128 KB are written to primary log and secondary log buffer and all batches equal or larger than 128

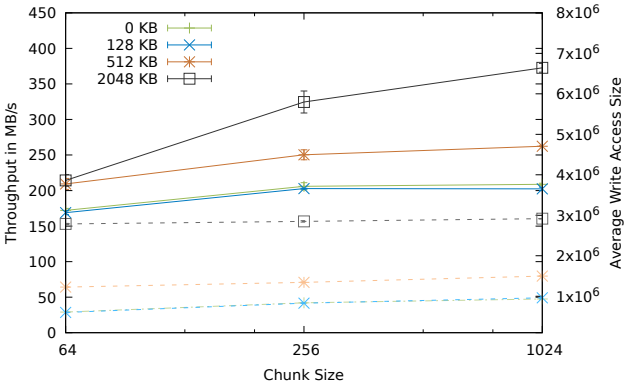

Figure 15: *Evaluation of the two-level logging with random distribution and varying secondary log buffer size, 200,000,000 chunks. Solid lines: throughput in MB/s, dashed lines: average write access size*

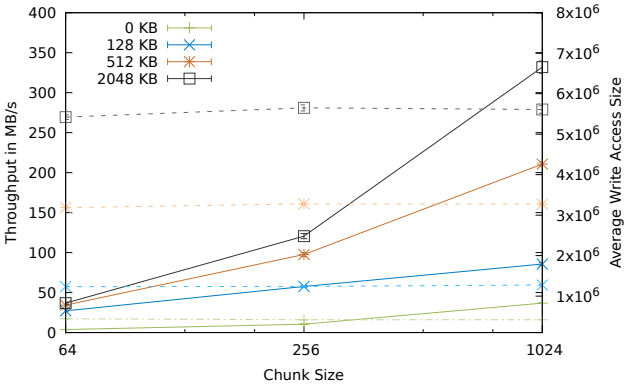

Figure 16: *Evaluation of the two-level logging with random distribution and varying secondary log buffer size, 2,000,000,000 chunks. Solid lines: throughput in MB/s, dashed lines: average write access size*

KB are directly written to the specific secondary log. With a size of 0, all log entries are flushed to secondary logs disabling the primary log and secondary log buffers.

Figure 15 shows that the two-level logging with secondary log buffer sizes larger than 512 KB increases the throughput by 20 to 25% for the random distribution and 200,000,000 64-byte chunks and up to 117% for 1024-byte chunks. This is due to the write accesses being much larger (dashed lines in Figure 15). However, using very large secondary log buffers increases the wear on the disk as many log entries are written twice (first to primary log, later to secondary log). Furthermore, the data processing is more time consuming than writing to disk in this scenario. This would not be the case for slower disks making smaller secondary log buffers more attractive. In this workload, using 128 KB secondary log buffers is as fast as disabling the two-level logging because the average batch size in the write buffer is considerably larger than 128 KB.

To decrease the log entry batch sizes, we repeated the test from above with 2,000,000,000 64-byte chunks (500,000,000 256-byte and 125,000,000 1024-byte chunks). Figure 16 shows that the performance advantage of utilizing the two-level logging increases with more chunks and thus smaller batch

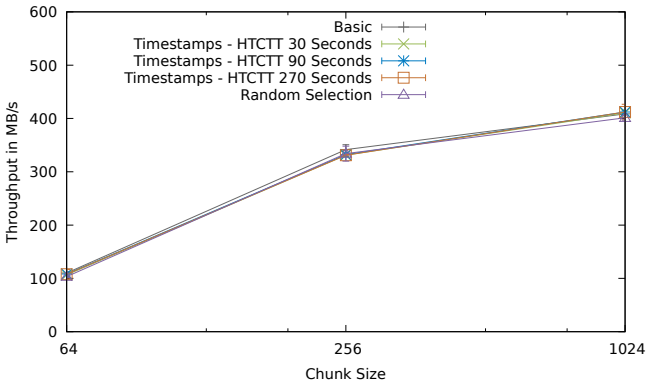

Figure 17: *Evaluation of the reorganization with timestamps. Sequential access distribution*

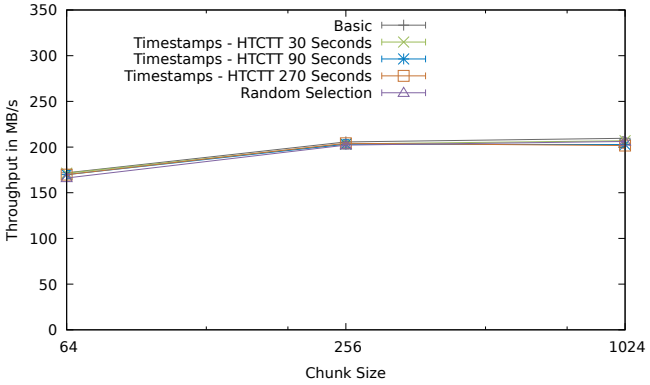

Figure 18: *Evaluation of the reorganization with timestamps. Random access distribution*

sizes when using the random distribution, as expected. With 128 KB secondary log buffers, the two-level logging improves the performance for 64-byte chunks by more than seven times in comparison to a normal logging scheme.

The random distribution is the worst case scenario regarding the decrease of batch sizes with increasing number of chunks because it scatters the accesses uniformly across all logs making aggregation less efficient with many chunks. The sequential distribution is unaffected by the number of chunks, the zipf and hotNcold distributions are less affected than the random distribution.

C. Timestamps

Figures 17 to 20 show the logging throughput with ongoing reorganization. In contrary to Section XI-B2, we used three different segment selection strategies: basic ($time\ since\ last\ reorganization\ or\ creation * utlization$), with timestamps to determine the average age of a segment ($age * utilization$) and random selection. We also varied the hot-to-cold transformation threshold (HTCTT) to study its impact on the performance.

Surprisingly, Figures 17 to 20 show that the segment selection has a negligible impact on the overall performance. For the sequential, random and hotNcold distributions all five selection strategy are equal regarding the logging throughput with ongoing reorganization. Only, for the zipf distribution, the

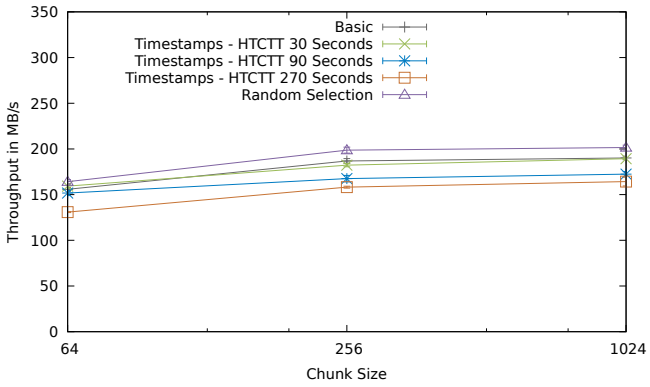

Figure 19: *Evaluation of the reorganization with timestamps. Zipf access distribution*

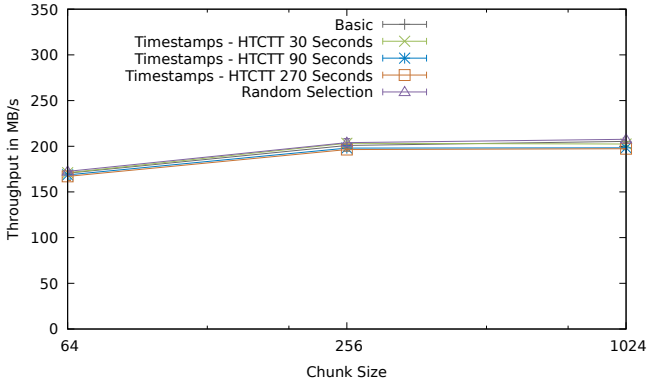

Figure 20: *Evaluation of the reorganization with timestamps. HotNCold access distribution*

throughput differs. However, the least elaborated strategy has the highest throughput in this scenario.

The timestamp selection has to be considerably better than the other strategies to outweigh the additional four bytes in the log entry headers. This is not the case here. For the random and hotNcold distribution the reorganization is not under pressure because the logging is restrained by the scattered access. Therefore, the segment selection cannot make a difference in this scenario. For the sequential distribution, the logging throughput is not restrained, but logs fill-up one after another, quickly triggering urgent requests for the current log. An urgent request initiates a reorganization of all segments in ascending order rendering the segment selection strategy irrelevant. For the zipf distribution, selecting older segments can be misleading because new segments of logs with a hotspot contain many already outdated versions of the very frequently updated hotspot. Hence, selecting a new segment for the reorganization can be better in this scenario. A low HTCTT has a positive affect on the segment selection as older segments appear much younger (older objects are left out for the age determination), in some cases even younger than a new segment.

D. Logging Remote Chunks

In this section, we evaluate the logging performance with chunks transferred over an InfiniBand network. We used the

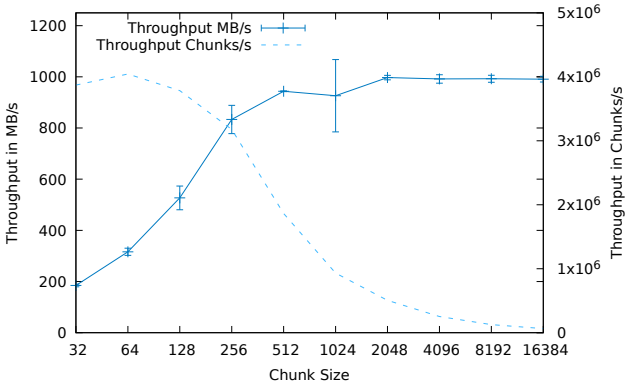

Figure 21: *Logging Throughput over InfiniBand Network*

O_DIRECT access, 8 MB segments, the two-level logging with 128 KB threshold and no timestamps. The checksums are used like in all other tests. The benchmark creates all chunks (up to 400,000,000), first. Then, the chunks are updated sequentially in batches of ten which are sent directly over the network to the backup server.

Figure 21 shows that no performance is lost when chunks are sent over the network instead of creating and logging them locally. DXRAM is able to update, sent, receive and log more than 4,000,000 64-byte chunks per second. The SSD is saturated with up to 512-byte chunks with a throughput of nearly 1 GB/s.

XII. Conclusions

In this report, we presented DXRAM's logging architecture in detail with focus on the data flow and the disk access methods, extending the papers [1] and [2]. Furthermore, we discussed the usage of timestamps to accurately calculate a segments age in order to improve the segment selection for the reorganization and we introduced copysets to DXRAM.

The evaluation shows the good performance of the logging and reorganization and demonstrates that DXRAM utilizes high throughput hardware like InfiniBand networks and nvme PCIe SSDs efficiently. DXRAM is able to log more than 4,000,000 64-byte chunks per second received over an InfiniBand network. Larger chunks, e.g., 512-byte chunks, can be logged at nearly 1 GB/s, saturating the PCI-e SSD. The reorganization is able to keep the utilization most times under 80% for realistic distributions (random, zipf and hotNcold) while maintaining a high logging throughput.

References

[1] K. Beineke, S. Nothaas, and M. Schoettner, "High throughput log-based replication for many small in-memory objects," in *IEEE 22nd International Conference on Parallel and Distributed Systems*, Dec. 2016, pp. 535–544.

[2] ——, "Fast parallel recovery of many small in-memory objects," in *IEEE 23rd International Conference on Parallel and Distributed Systems*, Dec. 2017, p. to appear.

[3] ——, "Dxram project on github," https://github.com/hhu-bsinfo/dxram, accessed: 2018-02-06.

[4] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," ser. SIGCOMM '01, 2001.

[5] H. Lu, Y. Y. Ng, and Z. Tian, "T-tree or b-tree: main memory database index structure revisited," in *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*, 2000.

[6] F. Klein, K. Beineke, and M. Schoettner, "Memory management for billions of small objects in a distributed in-memory storage," in *IEEE Cluster 2014*, Sep. 2014.

[7] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, pp. 7:1–7:55, Aug. 2015.

[8] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 92–105, Jan. 2010.

[9] B. B. V. Srinivasan, "Citrusleaf: A real-time nosql db which preserves acid," Aug. 2011.

[10] S. Sanfilippo and P. Noordhuis, "Redis," 2009.

[11] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, pp. 26–52, Feb. 1992.

[12] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan, "File system logging versus clustering: A performance comparison," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95, 1995, pp. 21–21.

[13] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 1–16.

[14] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum, "Copysets: Reducing the frequency of data loss in cloud storage," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 37–48.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 29–43.

[16] P. Londhe, S. Kumbhar, R. Sul, and A. Khadse, "Processing big data using hadoop framework," in *Proceedings of 4th SARC-IRF International Conference*, Apr. 2014, pp. 72–75.

[17] B. Caldwell, "Improving block-level efficiency with scsi-mq," *CoRR*, vol. abs/1504.07481, 2015.

[18] S. Chalasani, "On the Value of a Social Network," *ArXiv e-prints*, Dec. 2008.

[19] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, "Quickly generating billion-record synthetic databases," *SIGMOD Rec.*, vol. 23, pp. 243–252, May 1994.

Chapter 7.

Conclusion

This dissertation addressed two important research challenges for designing a fast, scalable and fault-tolerant distributed in-memory key-value store. The first two chapters focused on an open-source network library written in Java and integrated into the distributed in-memory key-value store DXRAM. In this context, we answered research questions regarding the network transport and communication model and presented our approach for achieving low-latency and high-throughput sending and receiving of (small) messages. The next two chapters presented a crash recovery concept for in-memory storages. In these chapters, we provided solutions answering research questions regarding the backup distribution and consistency, the log reorganization and the fast recovery of hundreds of millions of small data objects. Another chapter was dedicated to the optimizations of the crash recovery implementation to utilize high-speed hardware efficiently and to benefit from the proposed network subsystem which was developed after the logging and recovery.

The primary application domains of this thesis are big data analytics and large-scale interactive graph applications both demanding low-latency data access and high throughput for billions to trillions of mostly small data objects.

By providing fast parallel serialization of Java objects and an automatic connection management, the network subsystem can be used by many Java applications to send asynchronous messages or synchronous requests/responses. DXNet is optimized for high-throughput implementing an automatic message aggregation in lock-free data structures and an event-driven message receiving approach. Additionally, DXNet provides low-latency synchronous communication using a zero-copy and zero-allocation architecture. Typically, high throughput and low latency are contrary objectives. However, we developed different thread parking strategies to reduce latency for time-critical operations and relief the CPU when processing large batches. DXNet has a modular design and can be extended by transport implementations. In this thesis, we proposed an Ethernet transport and used an InfiniBand transport for evaluation purposes. EthDXNet, the Ethernet transport, is based on Java.nio and offers a batch operation interest handling to utilize Java.nio's Selector efficiently. Furthermore, we proposed a double-channel approach which allows concurrent bilateral connection establishment and sending/receiving of out-of-band data like flow control messages.

The evaluation of DXNet showed that message processing times are low (sub 300 ns) and throughput is high ($>$ 16 GByte/s). DXNet supports many application threads by providing constant processing times with up to hundred applications threads and increasing throughputs

with up to thousand remote threads on commodity servers in the cloud. With the InfiniBand transport, we measured average RTTs of under 10 µs and a duplex throughput of more than 10.4 GByte/s. We also showed that scalability of DXNet and EthDXNet is very good by achieving an aggregated throughput of 83 GByte/s and almost constant latencies in an all-to-all communication pattern with up to 64 nodes, connected with 5 GBit/s Ethernet (10 GBit/s Ethernet limited by SLA) in the cloud.

We proposed a crash recovery concept which stores replicas persistently on disks to mask server failures and power outages. The scalable range-based replica management distributes replicas to remote backup servers which store the replicas in logs on disk. We adopted the copyset idea from literature and integrated it into the replica management to reduce data loss probability for multiple simultaneous server crashes. By using the existing disks and a resource efficient logging approach, we do not need additional servers for backup as backup servers can also serve as storage servers and execute computations. Writing the replicas to logs improves the disk utilization, especially for small data objects. We further optimized disk utilization with the two-level logging approach providing fast persistence and high-throughput logging for varying application access patterns. Complemented by the novel fast and space-efficient epoch-based version management and a zero-copy direct disk access architecture, impact on fault-free execution is low and high-speed hardware, like InfiniBand interconnects and NVMe PCI-e SSDs, is saturated, even for small data objects. Furthermore, we proposed an efficient log cleaning approach which relies on different log and segment selection strategies.

The proposed backup architecture subdivides the data of one storage server into backup zones and scatters them to many backup servers. On backup servers, incoming replicas are sorted by backup zone. This enables a highly parallel recovery by reloading multiple backup zones concurrently on different backup servers which only read relevant data from disk (a single backup zone). Furthermore, invalid and outdated data can be excluded in the process as version information is stored on backup servers, as well. The reloading of a backup zone is executed by multiple threads locally to further reduce recovery times. Server failures are detected and recovery is coordinated based on a superpeer overlay. Our concept bypasses inquiring backup servers as superpeers know all backup zones and their backup servers of every storage server.

The evaluation showed that the proposed logging and reorganization approach is faster than the approaches implemented by the state-of-the-art systems RAMCloud, Aerospike and Redis. With InfiniBand and PCI-e SSD, more than 4,000,000 64-byte chunks per second can be logged on a single backup server. 512-byte chunks already saturate fast SSDs with a throughput of nearly 1 GByte/s. The reorganization keeps pace and holds the utilization most times under 80% for random, zipfian and hotNcold access distributions. A large-scale experiment in the Microsoft Azure cloud showed that a server with 500,000,000 64-byte objects can be recovered in under 2 seconds with 72 backup servers concurrently recovering fractions of the failed server's data. For small objects, our crash recovery approach outperforms RAMCloud's by a factor of nine without sacrificing performance for large objects. We also showed that fast consecutive recoveries can be executed even under high load (2,400 application threads).

7.1. Future Directions

Speaking of future directions, one has to consider hardware advances in the future. While the proposed concepts should scale well with increasing I/O bandwidth (e.g., 100 GBit/s InfiniBand) and processing power (like shown in Chapters 2 and 6), new technologies like byte-addressable non-volatile RAM (NVRAM) and RDMA could be interesting prospectively. NVRAM could not replace disk backups as NVRAM will most likely be too expensive to store backups as well. However, the logging architecture could be optimized for NVRAM as write buffers would be persistent. Furthermore, restarting after a cluster power outage could be considerably improved because servers would have all data available on restart without reading from disks. With falling prices of RAM, one could consider storing one replica in RAM providing instant availability for single server failures. In the case of multiple server failures, the proposed recovery concept would still be necessary. Implementing RDMA for DXNet could improve performance for suitable operations of Java applications. For the proposed crash recovery concept, we do not see applicable RDMA operations because writing to logs requires knowing the current write position and determining a monotonic version number and checksum for the replica which is slower if executed remotely.

The proposed online crash recovery concept requires backup servers having enough free memory to load the objects of a failed server into their memory. One can ensure this by continuously balancing the utilization of all storage and backup servers. Another approach is to stream recovered objects to another server which have enough free space to load them into memory. Neither of the two approaches has been implemented because of time constraints. This thesis does not cover a coordinated cold start after a data center power outage, either. However, we provide methods for recovering objects from files storing entire backup zones. The coordination and data migration has to be done in the future to sustain data center power outages. Elastic scaling is another feature to be considered for the future. Elastic scaling means automatically shutting down servers when the load is low and adding new servers if the load is high. For up- and downscaling, the replica distribution can be used to migrate entire backup zones reducing the metadata overhead and bypassing re-replication as backup servers do not have to be changed.

Because of time constraints, the crash recovery could not be evaluated with high-speed I/O devices. We expect that the recovery scales as well as the logging with increased I/O bandwidths. Furthermore, evaluating the crash recovery concept and the network subsystem with real-world graph applications, is something to be done in the future as applications are in development at the time this thesis is written.

7.2. Lessons Learned

DXRAM have become a large project with more than 100,000 lines of Java code (extended services, native code and scripts not included). Thus, implementing new concepts in DXRAM requires carefully designing, documenting and testing to obtain maintainable and extensible code. Most parts of DXRAM are optimized regarding performance and to handle small data objects with very low memory overhead. Therefore, much effort and time have been put into

maintaining and even increasing the high optimization level, which is difficult for a complex distributed system resulting in lengthy profiling and debugging sessions. Occasionally, problems were not even software-related. For example, the performance was compromised by a low-budget Ethernet switch whose actual switching capacity was significantly lower than specified (switch was replaced). Another example: logging performance was subpar because the garbage collector on the SSD affected the write throughput when experiments were executed consecutively. Removing the log files and discarding unused blocks (by calling fstrim) before executing the next experiment solved this problem. The configuration of the operating system can also impact the performance significantly, e.g., socket buffer sizes or page-cache thresholds.

Many computer scientists, especially in the high-performance computing area, have reservations regarding object-oriented, generic programming languages like Java. In this thesis, we showed that the concepts and the code quality are more important than the programming language, especially for distributed programs. All proposed concepts implemented in Java are highly concurrent and provide low latency and high throughput. Evaluations showed that our Java-based distributed in-memory key-value store was faster in all tested scenarios than, for example, RAMCloud and Aerospike, both written in C/C++. Another critical performance aspect is to adjust concepts to the primary application domain which in this thesis are applications with billions to trillions of small data objects.

Part I.

Appendix

Chapter 8.

Appendix

8.1. DXRAM - Additional Information

Configuration

All applications, services and components of the DXRAM ecosystem can be configured using JSON configuration files. We use the Google's gson library [1] which enables automatic serialization of Java objects to JSON files and vice versa. Every DXRAM component and service is accompanied by a configuration class. In this configuration class, every configurable value is marked with the annotation @Expose at declaration and the initialization value is used as default value. Values are read during runtime through getters. Furthermore, every configuration class contains a method to verify the configuration values (i.e., print a warning if a value is set impractically or abort the DXRAM initialization if a value is not permitted).

During the initialization of DXRAM, the DXRAM engine creates a DXRAM context that includes all configuration classes of all components and services. The DXRAM context is passed to all services/components and the service-/component-specific configuration is available at all times within the instance.

On first startup of DXRAM, all default configuration values are written into a file. Based on this file, the user can manipulate the configuration values for the next start of DXRAM. At last, configuration values can be passed as JVM arguments.

More Extended Services

DXTerm: DXTerm is a DXRAM application providing a CLI for DXRAM. DXTerm offers a set of commands for configuring, debugging, monitoring and using DXRAM services (see Table 8.1). We use the jline library [49] which provides similar functionality as GNU readline. Thus, the CLI is akin to most modern shells.

The application is loaded automatically on starting of DXRAM and one can connect to the application at any time by executing an accompanying script. At the time of writing this thesis, a web user interface is developed using a Representational State Transfer (REST) API based

on the Hypertext Transfer Protocol (HTTP). The web interface will provide comfortable access to DXTerm's functionality.

DXMonitor: DXMonitor is a monitoring and management service for DXRAM, developed by Burak Agkül. The monitoring service collects live information of the server's CPU, memory, disk and network by reading from the ProcFS [82]. Furthermore, information from the JVM (e.g., garbage collection utilization) is collected and information about DXRAM services and components through an extended API. Different metrics (average, min, max, median and percentiles) are used to calculate the monitoring data on data servers. The monitoring data is sent to metadata servers periodically and the metadata servers can use the information of all data servers to optimize the performance of DXRAM, for instance, by triggering migrations in order to balance the load. The monitoring data is also written to CSV files on disk by dedicated threads which are used to visualize the data in a monitoring web console. At last, the monitoring information can be gathered using the monitoring command of DXTerm which is available if DXMonitor was started, only.

cdepl: cdepl is a collection of scripts creating a framework to simplify the deployment of distributed applications to different (Linux) cluster setups. It creates an abstraction layer to the target cluster system for the application to deploy. This enables clear separation of the actual hardware to deploy to and the application getting deployed. Different cluster (type) implementations map to specific cluster setups with their dedicated environments. For example, the localhost type maps the abstraction layer to the current machine for quick testing or simple debugging tasks. The simple cluster type deploys to an arbitrary cluster setup by providing a list of hostnames. Further types handle environment specific features for each cluster system. Applications are implemented in separate modules, as well. With every application offering different features and requiring different ways to control it, there is no unified abstraction layer for them. However, creating abstractions of tasks makes it easier to write small and powerful deploy scripts.

cdepl is entirely implemented as bash scripts and uses common Linux utilities, only, to avoid further dependencies. In addition to the already mentioned cluster types, we support special cluster types like the Microsoft Azure cloud, hilbert (the HPC of the Heinrich-Heine university) and our private cluster. Additionally, applications like DXRAM, DXNet, Aerospike, RAMCloud, Redis and ZooKeeper are supported. Furthermore, the YCSB can be started with cdepl using DXRAM or RAMCloud as backend storage. Support for further applications or clusters can be added easily.

When starting DXRAM or DXNet with cdepl, the configuration files (see Section 8.1) are generated automatically according to the deployment. The configuration is either based on default configurations of all services and components or a previously generated (and possibly modified) configuration file allowing user-specific adaptations. cdepl also allows remote debugging and profiling of DXRAM.

Core Services

In the following, we give a brief overview of all DXRAM services available to the time of writing this thesis.

- Application: Run DXApp jar packages as a per application dedicated thread (see Section 1.3.1).
- Boot: Expose own node ID, node ID mappings, capabilities, etc. (see Section 8.1).
- Chunk: Key-value storage interaction; Create, get and put chunks/data structures (see Section 1.3.4). Uses the submodule SOH. To remove chunks, the remove service (see below) is used.
- Compute: Run and coordinate task-based computations on data servers (see Section 1.3.2).
- Job: Enqueue new jobs either locally or to remote nodes running the job service (see Section 1.3.2).
- Lock: Lock either local or remote chunks (see Section 1.3.4).
- Log: Write backup data to (remote) disks (see Section 1.3.5).
- Logger: Access the local or a remote (text) logger (e.g., set logger level). We use the log4j library for message logging in six different levels (TRACE, DEBUG, INFO, WARN, ERROR, DISABLED).
- Lookup: Access to the (remote) superpeer overlay, lookup tree, metadata, etc. (see Section 1.3.3).
- Migration: Migrate chunk(s) from one peer to another (see Section 1.3.4).
- Nameservice: Stores name to chunk ID mappings on metadata servers (see Section 1.3.3).
- Network: Send and receive messages/requests. Uses submodule DXNet(see Section 1.3.6).
- Recovery: Coordinate the local recovery of backup zones (see Section 1.3.5).
- Remove: Remove chunks from memory in a dedicated dispatch thread to avoid remote deadlocking (see Section 1.3.4).
- Statistics: Access to statistics collected in various components and services. Statistics are used for debugging and can be disabled to increase performance.
- Synchronization: Barrier synchronization for computations on peers (see Section 1.3.3).
- Temporary Storage: Small, auxiliary and chunk-store-independent scratch pad on metadata servers (see Section 8.1).

Core Components

In the following, we give a brief overview of all DXRAM components available to the time of writing this thesis.

- Application: DXApp package management for applications running on DXRAM servers (see Section 1.3.1).
- Backup: Management of backup ranges and backup tree (see Section 1.3.5).
- Boot: Node bootstrapping, node ID assignment, node mappings and handling of capabilities (see Section 8.1).
- Chunk: Access to local memory for DXRAM management data (see Section 1.3.4).
- Event: Event signaling and handling system (see Section 8.1).
- Failure: Server failure handling (see Section 8.1).
- Job: System running worker threads executing queued jobs (see Section 1.3.2).
- Lock: Storage for active locks on locally stored chunks (see Section 1.3.4).
- Log: Logging (persistent data backup) with access to disk (see Section 1.3.5).
- Lookup: Overlay management and lookup cache tree (see Section 1.3.3).
- Memory: Memory allocator and manager with access to native memory (see Section 1.3.4).
- Nameservice: Naming index structure (see Section 1.3.3).
- Network: Access to DXNet (see Section 1.3.6).

Temporary Storage

DXRAM provides a small volatile storage on superpeers for putting coordination information, debug information or temporary results. The storage is replicated to three superpeers but is not stored on disk. Thus, after a power failure, the data is lost and cannot be recovered. The storage should be used for depositing temporary data like a job ID or a file path, only. Every information stored in the temporary storage must have a unique ID. Data with the same ID is overwritten and the order of concurrent access is non-deterministic. Thus, the storage cannot be used for solving consensus.

The temporary storage will be replaced by a storage implementing the Raft consensus algorithm [83] in the future. This avoids using ZooKeeper [47] for bootstrapping and extending the set of

participating servers which requires consensus (see Section 8.1).

Bootstrapping

The bootstrapping of DXRAM was part of the master thesis of Kevin Beineke [7].

To start a DXRAM server, a configuration file has to be passed to the DXRAM engine. The configuration file defines all configurable values of all components and services and also describes the role, address and capabilities of all participating servers. The first server in the configuration must be a superpeer and is the bootstrap server for every other joining server. A bootstrap server is necessary to avoid partitioning the superpeer overlay on startup. If the bootstrap server fails, another server is selected and stored in ZooKeeper. Every joining server parses the configuration file and adds all servers to an array, called the nodes configuration. The nodes configuration stores the role (peer or superpeer), the address (e.g., IP and port), the location in the cluster (rack and switch; used for location-aware replica placement), whether the server is online or offline and the capabilities (e.g., whether the peer is available for backup) for every known server. Furthermore, we distinguish if a server was registered in the configuration file or added later.

While parsing the configuration file, the node IDs of all servers are determined. Every server determines identical node IDs for all servers by using the same algorithm (CRC16 with equal seed). To speed-up node ID determination, we use a bloom filter to detect collisions (same node ID for two servers). This approach is much faster than using, for instance, a consensus algorithm for propagating a servers node ID but requires that every server has the same configuration file. Generally, DXRAM is deployed with a dedicated deployment script (cdepl; see Section 8.1) which takes care of creating, adapting and copying the configuration file.

The set of DXRAM servers can be extended during runtime. These servers are not required to be included in the configuration file. Joining servers use a free node ID from a failed server, which are stored in ZooKeeper, or, if not available, creates a new node ID by hashing its address. Subsequently, the new server registers the used node ID in ZooKeeper (might have to generate a new one if the node ID is already in use). Every joining server also iterates the list of failed and new servers on startup to update its nodes configuration.

The initial filling of the nodes configuration is important to enable establishing a connection (on demand) to all participating servers. An event-based propagation protocol maintains the state of a server (online or offline). Every server informs its responsible superpeer when the startup is completed. Then, the superpeer sends the information to all other superpeers. The other superpeers forward the message to all their assigned peers. On every server, the receipt of the message fires a join event which is passed to all components/services registered for the type of events. This includes the boot service which updates the node configuration (set server state to online and update capabilities if necessary). If a server fails, a similar protocol is used which is started by the superpeer detecting the failure. The status information and capabilities can be used by every component/service, extended service and application. For example, the backup component uses the nodes configuration to select backup servers.

DXNet - Transport Interface

The following methods must be implemented for every transport [18]:

- Setup connection
- Close and cleanup connection
- Signal to send data available in the ORB of a connection (callback)
- Pull data from the ORB and send it
- Push received raw data/buffer to the IBQ

Table 8.1.: DXTerm's Command Set

Command	Parameters	Description
barrieralloc	number of peers	create a new barrier for synchronization of multiple data servers
barrierfree	barrier ID	free an allocated barrier
barriersignon	barrier ID	sign on to an allocated barrier
barriersize	barrier ID, number of peers	change the number of servers to wait for
barrierstatus	barrier ID	get the current status of a barrier
chunkcreate	size, node ID	create a chunk on a remote node
chunkdump	chunk ID, file	dump the contents of a chunk to a file
chunkget	chunk ID, format	get a chunk
chunklist	node ID	get a list of chunk ID ranges
chunklock	chunk ID	lock a chunk
chunklocklist	node ID	get a list of all locked chunks
chunkmigrate	chunk ID, source node ID, destination node ID	migrate a chunk
chunkput	chunk ID, data, format	put data into a chunk
chunkremove	chunk ID	remove an existing chunk
chunkremoverange	first chunk ID, last chunk ID	remove a range of existing chunks
chunkstatus	node ID	get the status of the chunk service
chunkunlock	chunk ID	unlock a previously locked chunk
compgrpls		get a list of available compute groups
compgrpstatus	group ID	get the current status of a compute group
comptask	task, group ID	submit a task to a compute group
comptaskscript	file, group ID	submit a list of tasks loaded from a file
loggerlevel	level, node ID	change the output level of the logger
loginfo	node ID	print the log utilizations
lookuptree	node ID	print the lookup tree of the specified node
memdump	node ID, file	create a full memory dump
metadata	node ID	print a summary of the metadata server's data
nameget	name	get a nameservice entry
namelist		get a list of all nameservice entries
namereg	chunk ID, name	register a chunk in the nameservice
nodeinfo	node ID	get information about a node
nodelist	role	list all available servers with given role
nodeshutdown	node ID	shutdown a server
nodewait	number of servers	wait for a minimum number of servers
statsprint		print all statistics
tmpcreate	ID, size	create a temporary storage
tmpget	ID, format	get a chunk from temporary storage
tmpput	ID, data, format	put a chunk into the temporary storage
tmpremove	ID	remove a chunk from temporary storage
tmpstatus		get the status of the temporary storage

8.2. Asynchronous Logging and Fast Recovery for a Large-Scale Distributed In-Memory Storage

This chapter summarizes the contributions and gives a copy of our paper [10].

Kevin Beineke, Florian Klein and Michael Schöttner. "Asynchronous logging and fast recovery for a large-scale distributed in-memory storage". In: Plödereder, E., Grunske, L., Schneider, E. & Ull, D. (Hrsg.), Informatik 2014. Bonn: Gesellschaft für Informatik e.V.. pp. 1797-1810

8.2.1. Paper Summary

This workshop paper describes the initial ideas and concepts of the backup and recovery mechanisms. The mostly revised concepts are described in more detail in Chapter 4 and 5.

Asynchronous Logging and Fast Recovery for a Large-Scale Distributed In-Memory Storage

Kevin Beineke, Florian Klein,
Michael Schöttner
Institut für Informatik, Heinrich-Heine-Universität Düsseldorf,
Universitätsstr. 1, 40225 Düsseldorf, Germany
E-Mail: Kevin.Beineke@uni-duesseldorf.de

Abstract: Large-scale interactive applications and online graph analytic processing require very fast data access to many small data objects. DXRAM addresses these challenges by keeping all data always in memory of potentially many nodes aggregated in a data center. Data loss in case of node failures is prevented by an asynchronous logging on flash disks. In this paper we present the architecture of a novel logging service designed to support billions of small data objects, flash disk characteristics and fast node recovery. The latter is challenging if 32-64 GB of in-memory data of a failed node needs to be recovered in seconds from the logs.

1 Introduction

Large-scale interactive applications and online graph computations must often manage many billions of small data objects. Facebook for example supports more than one billion users by keeping around 150 TB of data in more than 1,000 memcached servers [ORS+11a]. Around 70% of these objects are smaller than 64 byte [NFG+13] and as a result each memcached server stores a few hundred million of objects. The sheer amount of objects and the small data sizes can be found in many other online graph applications, e.g. [SWW+12]. Another important aspect are the access patterns of these applications where reads dominate over writes [BAC+13], [AXF+12]. And depending on the type of data there are also deletes and updates but both less frequent than reads.

DXRAM is a distributed in-memory system aiming at managing billions of small data objects [KS13]. The core implements a key-value data model for binary data with basic get and put operations. The system design is open to execute algorithms on storage nodes or to run DXRAM as an ultra-fast back-end storage. A super-peer overlay is used for scalable node lookup and a paging-like translation scheme for efficiently mapping global unique keys to local virtual addresses. On top of the core, we plan extended data services, e.g. richer data models and naming/indexing services.
In this paper we present the design of our novel asynchronous logging approach, which is storing replicas of objects on remote flash disks to prevent data loss potentially caused by software and node failures. Replication in memory is not an option as memory is very expensive and power outages may draw down many nodes resulting in data losses. The

logging is designed carefully, taking into account SSD characteristics, to maximize write performance. The latter requires basically sequential writes, which results in a sequential log. This however requires to provide a cleaning mechanism similar to the work published in log-structured file systems [RO92].
Another aspect is that we design the log in a way to allow very fast recovery of nodes states (32-64 GB of data), which is very challenging [ORS+11a]. Fast recovery will be even more challenging as we aim at supporting billions of small data objects.

This paper is structured as follows. In section 2 we give a short overview of the DXRAM system including the global meta-data and in-memory data management. Section 3 presents the logging approach, followed by section 4 and 5 where we describe the recovery process and the cleaning of logs. Related work is discussed in section 6 followed by the conclusions and an outlook on future work.

2 DXRAM Architecture Overview

DXRAM is a distributed in-memory storage system designed for the management of binary data in data center environments. We aim at supporting billions of small data objects (16-64 byte) which is for example required for managing very large graphs, e.g. social networks. By keeping all data always in RAM we can provide low-latency data access for all data. Furthermore, by providing transparent persistence we relieve programers from synchronizing caches with secondary storage.

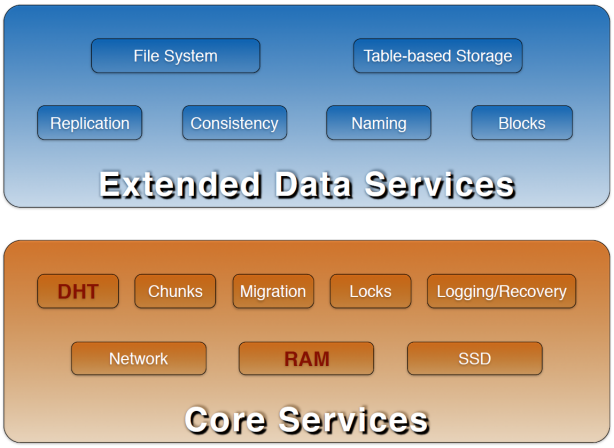

Figure 1: DXRAM architecture

Figure 1 shows the overall architecture of DXRAM. Extended data services include general services and extended data-models built upon core services. Not all extended services shown in the figure are implemented but we designed the core to be flexible allowing for example to support different data models as needed without requiring to re-implement

everything from scratch.

Core services provide functionality for the management, storage and transfer of chunks (key-value tuples). One of the main objectives in core services is to keep the functionality and the interface for high layers as compact as possible. The minimal interface for chunks includes following functions *create*, *delete*, *get*, *put* and *lock*.

The DXRAM core implements a key-value data model where a tuple is called chunk. A chunk has a 64 bit globally unique chunk ID (CID) followed by the value (raw bytes). The CID is composed of two parts. The first 16 bit is the node ID of the creating node (NID_C). The remaining 48 bit is a locally unique value, named LID, which is incremented during each local chunk creation. This results in a sequential CID generation scheme on every node. The key size is configurable but with the described numbers we can address 65,536 nodes each storing up to 280 trillion chunks. We decided to use this scheme in favor of user-defined keys to avoid hashing allowing us to keep IDs compact, to support locality-based access patterns and to avoid adjusting hash functions in case of nodes scaling up and down.
But the sequential ID generation is not a constraint for the applications. On the one hand most applications which use databases as persistent storage access data through auto-incremented row IDs similar to our LID. On the other hand DXRAM provides a name service to address a CID using a user-defined key. The super-peers store the mapping of keys and CIDs in a patricia-trie structure. The intention is that not each single object needs a user-defined key but only a subset, e.g. the user records in a social network.

Chunks have variable sizes defined during their creation always stored en bloc. The basic get and put operations on chunks read and update always full chunks. Because RAM is expensive we replicate chunks for fault tolerance on multiple backup nodes only in flash memory.

2.1 Global Meta-Data Management

A super-peer overlay is used for implementing a custom DHT allowing fast node lookup of chunks. Because of the high-speed network and the limited number of super-peers, e.g. 8-10% of all nodes we decided to keep them knowing each other in contrast to traditional internet-scale DHTs like Chord, CAN, etc. This in turn allows lookups with O(1) time complexity. In addition peers cache node-lookup results reducing load on super-peers. Meta-data is replicated on the neighboring super-peers to mask super-peer failures. However, we do not need a high replication factor, which would be very costly, as meta-data can be dynamically re-constructed.
In a controlled environment like a data center we expect only small node churn caused by failures. Of course a power outage may kill all our nodes and will require to restore all data from all logs, which will take considerable recovery time; but we expect this disaster to be very seldom. Otherwise we expect controlled up and down scaling of nodes as needed dynamically using the super-peer overlay. If we reach a defined threshold a new super-peer is promoted or respectively demoted. In both cases the hashing function for the DHT does

not need to be adjusted thus not requiring to re-hash entries (like for a traditional DHT).

The super-peer overlay can aggregate the meta-data of chunks because of the sequential CID creation and bundles multiple CIDs to a CID Range (start, end) together with the NID_O (o: actual owner). Note, the actual owner of an object may be different to the creator if an object has been migrated, e.g. because of load-balancing reasons.
For example, if we have 1,024 chunks with CIDs 1 to 1024 created on one node, then we have only one entry for this CID range on the associated super-peer. In addition to avoid gaps in the CID ranges through deletions of chunks, we reuse free CIDs for new chunks. As super-peers have to map billions of CIDs from all their associated peers this compact representation is very space efficient.

As mentioned above DXRAM also supports chunk migrations to relieve load on peers storing possibly several hot spots. For example in the context of social networks a hot spot can be a famous artist whose profile is very often visited (some artists have up to 40 million friends in Facebook [SWL13]). Although migration solves the load problem it results in a range split of the global ID range on the super-peer responsible for the migrated object. Still this is not a problem for the global meta-data management because we expect that such chunk migrations are seldom as we do not expect billions of hot spots. A fast lookup CIDs in chunk ranges is ensured by implementing a B-tree-like data structured described in [KS13].

Beside meta-data management, super-peers are also responsible for recovery coordination of failed nodes which is discussed in chapter 4 and for load monitoring which is beyond the scope of this paper.

2.2 In-memory Data Management

We aim at managing up to one billion of small objects on one node, which is challenging especially concerning the mapping of global IDs to local virtual addresses as well as the local memory management. Although both topics are well studied in literature the sheer amount of objects raise new challenges regarding space efficiency and access times.

Many systems use hash tables for the translation of global IDs to local addresses which tend to waste memory or get slow when the load factor increases (causing collisions). In contrast we implemented a paging-like translation scheme which is fast and space efficient.

Furthermore, most memory allocators are not optimized for many small allocations. On the one hand they align memory for cache performance reasons (4/8 byte granularity) and on the other hand they append a header (4 - 64 byte) depending on programming language and runtime system. Obviously, the overhead is too large, e.g. for one billion objects, each with a 16 byte header, we would need 16 GB RAM just for the headers. Because memory is expensive we have decided to design a memory management trading cache performance for minimizing internal fragmentation. Cache penalties are not critical for DXRAM (when used as back-end storage service) as the time for client operation requests is dominated by network transmissions.

The global ID mapping and the local memory management are described in a paper, currently submitted to another conference.

3 Organization of the Log

As mentioned before DXRAM keeps all data always in memory. In order to address node failures, power outages and not to burden the programmer with persistence management on secondary storage we introduce an asynchronous logging service described in this section. Each object is replicated on a configurable number of remote nodes called backup nodes. These backups are stored in a log optimized for SSDs (, which could also be applied to traditional disks). During fault-free execution only write operations of updates are executed on the log but no reads. In case of a recovery request, when a node crashed, the full log is read (described later in more detail). A fast recovery is only possible if the state of one node is spread over many backup nodes. This allows to aggregate SSD and network bandwidths during recovery [ORS+11a]. Therefore, we introduce backup zones, similar to the segments in RAMcloud, although we do not use static sizes but allow dynamic size adaption. Each node providing RAM is also providing SSD backup-capacity.

Before we start with the presentation of the log service we discuss the characteristics of SSDs. In contrast to traditional disks, SSDs read and write clustered pages, each 4 KB and each on another NAND flash memory accessed via multiple channels. This means, writing one byte results in writing at least 4 KB. Obviously, as we aim at supporting many small data objects (16 - 64 byte), we need to introduce some buffering for bundling write accesses. To benefit from internal parallelism of SSDs it is even better to write multiple pages at once [MKC+12]. Furthermore, SSDs cannot re-write a page, but instead have to delete the block (64 to 128 pages) first. The write access will then be done on a page the SSD controller chooses. To manage the complexity of that fact the SSD controller uses a specific garbage collection and flash translation layer to hide the characteristics of flash memory. This results in a much higher bandwidth while writing sequentially than randomly [MKC+12]. Finally, two more aspects to be taken into consideration are to avoid delete operations whenever possible because of the large erase block granularity (depending on model, e.g. 256 KB - 2,048 KB) and not to mix read and write accesses as both can harm each other [CKZ09]. The latter is no problem as read and write operations are anyway separated (fault-free execution and recovery), see above.

3.1 Primary Log

DXRAM organizes logs into two levels, one primary log and several secondary logs, one for each node requesting backups. The objective of the primary log is to store incoming backup requests as soon as possible on SSD to avoid data loss caused by software or hardware failures. The primary log is organized as a ring buffer similar to SpriteFS' [RO92] and all incoming backup requests (from any node) are bundled together in the primary log.

In contrast to log-structured file-systems the log is never read during fault-free execution. Thus there is no need to store meta-data of the log in memory. However, the log is self describing, each entry has a header with a length field, NID and LID for identifying the object and optionally a CRC checksum, and is fully read-in during node recovery.

In order to avoid many small write accesses to the primary log, all incoming backup requests are bundled in a write buffer. The buffer itself is organized as a ring buffer and the access is performed using several producer and one consumer thread. The producer threads decouple network threads processing backup requests, however, the node requesting a backup may force a synchronous operation, if requested by the above application. This is optional in case of critical updates, under control by the applications.

It is important to note that the buffer will be filled by backup requests from potentially many nodes, which partially backup their state on a backup node. So, it is likely that there will soon be enough data in the buffer to flush one page out to the primary log or even multiples of a page, depending on the update frequency in the overall system. However, we set a configurable timeout ($<$1s) for the consumer thread to flush out data to SSD. The idea is to avoid an increased data-loss probability in case of very low update frequency.

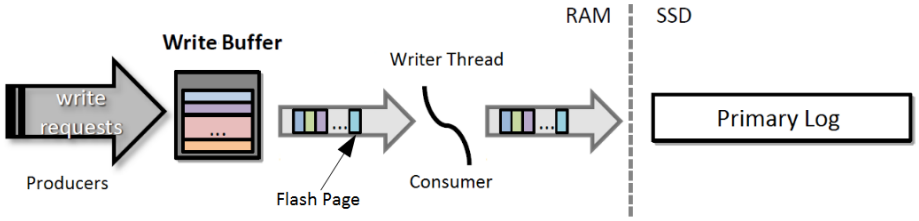

Figure 2: Interaction between write buffer and primary log

Obviously, there is a small time interval where an update is vulnerable to a power outage or failure of all nodes storing the update in memory. Of course in the worst case scenario the full data center could go down by a power outage. The latter is very unlikely, whereas the first may happen. As a consequence we provide applications a sync command to enforce write through for backups of critical updates.

Right now we would have to read in and analyze the full primary log during a recovery requests to detect log entries of the crashed node and skip all other entries. This is obviously inefficient, which in turn lead us to establishing the secondary log.

3.2 Secondary Logs

Each node distributes its state over numerous backup nodes, e.g. 64 GB RAM of one node is distributed over 100 or more backup nodes. Thus each node will service as storage node and as a backup node for potentially many other nodes. To speed up recovery we decided to sort backups into one separate secondary log for each node. Thus we can avoid reading in one large log storing backups from many nodes and analyzing which entries are needed

for a recovery.

After having written a backup request into the primary log, the data is also copied (in memory) into the secondary log buffer using the NID, see Figure 3. The secondary log buffer array grows dynamically to provide one write buffer for each node whose backups are to be stored. The reason for buffering writes to secondary logs is again to bundle them into at least one 4 KB page (similar to the primary log).

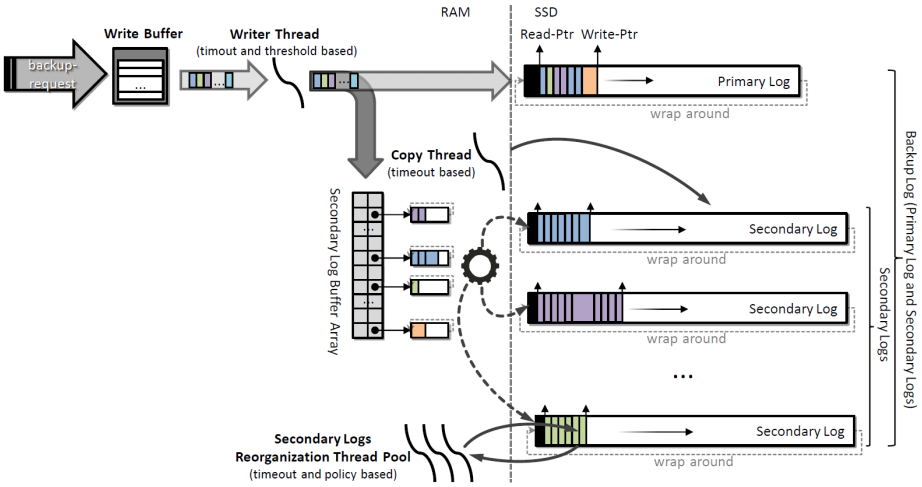

Figure 3: Log overview

It is important to note that the secondary log buffer does not introduce any risk of loosing data because all backups are stored in the primary log on SSD. However, the write buffers are flushed periodically (less frequently than the primary write buffer) by the copy thread. The benefits are threefold: First, the primary log can remove all objects that are written to the corresponding secondary logs by adjusting the read pointer only. Second, the reorganization of secondary logs is more efficient with all objects in log (more in chapter 5). And third, the recovery process is faster if no or only a few objects in the primary log have to be processed (more in chapter 4).

In case of frequent updates from a node, backups may skip the primary log and secondary log buffers and are directly copied from the primary write buffer into the secondary log.

The secondary logs also allows us to reduce the object header per entry. First, there is obviously no need to store the NID_O in the headers because every object in a secondary log has the same NID_O. Furthermore for consecutive object IDs it is sufficient to only store the full LID of the first object and a flag for successor entries. We expect this case to occur quite often as we allocate objects with consecutive IDs on each node and expect updates to be rather seldom. Furthermore, during recovery we have to read in the full secondary log and can reconstruct consecutive IDs easily.

4 Recovery

In large-scale clusters node failures are a rule, not an exception. Therefore, DXRAM keeps the states of nodes in a log spread over backup nodes, see Section 3. Below we describe how the recovery works for a failed node.

Node failures are detected using timeouts in case of data/backup requests by any node. In addition we use the super-peer overlay (described in Section 2), to establish a hierarchical heart beat protocol between each peer and its associated super-peer. Any peer suspecting a peer failure informs the responsible super-peer, which may wait for the next heart beat. The recovery of a failed node is controlled by its associated super-peer, see Figure 4. This will allow distributing recovery control on different super-peers if several peers fail.

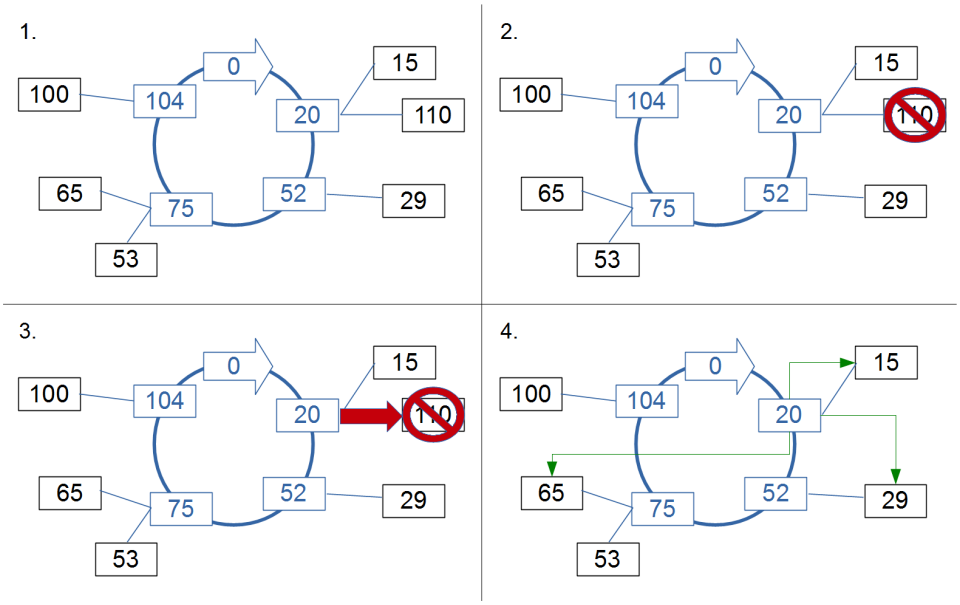

Figure 4: Recovery process: 1. Super-peer overlay with four super-peers (20, 52, 75, 104) and six peers (15, 110, 29, 53, 65, 100) 2. Failure of peer 110 3. Failure detection 4. Distribution of recovery messages

The first step of the super-peer during recovery is to determine backup nodes of the failed peer. As mentioned in Section 2, super-peers provide fast node lookup using CID ranges. These CID range entries do also include a list of backup nodes, typically 2-3 backup nodes. Thus the super-peer can easily determine, which backup nodes to contact to recover the full state of a crashed node. If objects have been migrated they belong to the node state of the new node and no longer to initial creator. As object migrations are also stored in the super-peers (resulting in a CID range split) this does not cause any problems during recovery.

If a super-peer crashes it is recovered by the super-peer overlay by a neighbor super-

peer, which is holding replicas of the meta-data. In order to not burden the system much by meta-data replication on super-peers, we only have 1-3 replica for each super-peer state. As we use dedicated nodes as super-peers, we expect enough memory for such a configuration. In rare cases several super-peers may crash simultaneously, resulting in meta-data loss. Still this disaster scenario will not lead to data loss and after the super-peer overlay structure has been repaired, the new super-peers will recover their meta-data through a multicast to gather CID ranges from their associated peers, which will also include backup nodes and migrated chunks. The chunk IDs of the latter cannot be re-used locally and thus we can use the entry in the paging-like map tables for storing the node where the chunk has been migrated to. This will however also require that if a chunk is migrated from the creator to another node and from there again to a new node, that the creator will be kept informed. This approach is also used after a full power outage, which would require to rebuilt all super-peers, which can be done in parallel like described above.

If a super-peer has gathered all backup nodes to be contacted it will always start with backup 1 and if this one is unavailable go to backup 2 and so on. The idea behind this sequential backup node selection is supported by the strict backup update order during fault-free execution, which is performed asynchronously for performance reasons. The latter ensures that the newest versions are always found in backup 1 and if unavailable we got for 2, where we still hope to find the most recent versions. Depending on the failure situation we might lose the newest version of an object because the object holder crashed and backup 1 crashed too and the other backups did not get updated because of message loss on the network. Obviously, this is a rather seldom event in a data center, it still could occur. In that situation we plan to recover some older version and inform the application. Again we want to point out that the core API will include a sync command, which allows applications to force synchronous backups for critical operations. But of course the overall system performance would be burdened if an application programmer will place after each put a sync.

After a backup node receives a recovery message for a given NID, it first flushes its write buffer and the corresponding secondary write buffer. This will ensure that all backup copies of the failed node are in its secondary log on SSD. It now depends on the configuration regarding the number of backup nodes, which will affect the size of the secondary log, e.g. if we have storage nodes with 32 GB RAM and distribute their state on 100 backup nodes, each backup node will potentially store 320 MB of data in its secondary log. To avoid pressure on the re-organization (described in the following section), we use at least double size for the secondary log, for the example above resulting in 640 MB for the secondary log per node. This is no problem as storage capacity on SSDs is increasing and getting cheaper, so we can expect at least 512 GB, which would allow each node to store logs for around 800 storage nodes. Depending on the free memory of the backup node, it might be possible to read in the full secondary log at once and analyze it in memory with multiple threads benefiting of multiple cores. Analyzing is necessary to determine the newest versions and also deleted chunks. Otherwise, if the backup node does not have enough memory to load the full secondary log, it needs to analyze it step by step, e.g. 16 MB per step.

Assuming all backup nodes have enough memory, they all could recover the secondary

log in their memories, could inform the super-peer, and could from then on serve chunk requests. This is the fastest possible recovery as this approach does run in parallel without any data transfers over the network. However, data locality is not preserved, as the state of the recovered node is now spread over numerous or many nodes. Therefore, it is planned to asynchronously move data from backup nodes to a fresh node to rebuilt the crashed node. In contrast to existing solutions, we plan to do this asynchronously, being able to serve requests, while we move larger chunk sets to the fresh node. So far we have implemented the local recovery, loading the full secondary log into the memory of the backup node and run a multi-threaded analysis.

5 Reorganization

The write buffers and the primary log are flushed periodically or when running full (beyond a given threshold), so there is no need for reorganizing them. However, the secondary logs are contiguously filled with update and delete log entries appended to the end of the log. For each chunk creation, update, and deletion we create one log entry.
Obviously, there is a strong need for reorganization in order to free space of outdated and deleted log entries, at latest when the log size reaches a predefined threshold. This is a known issue in log-structured file systems, which also benefit of the sequential high-speed write operations but have to spend considerable efforts for cleaning the log when running out of space.

Our current prototype uses a rather simple reorganization scheme. If the log size exceeds 75% of the overall capacity we trigger a log reorganization. The latter reads in a full secondary log and runs a multi-threaded cleaning using a thread pool with a work stealing mechanism. The cleaner threads analyze the full log for valid, deleted, and outdated entries. Because of the strict sequential write order we know that newer version are stored in the log after older ones.

While this basic in-memory solution performs well, we are currently studying incremental and more space-efficient approaches avoiding to load the full secondary log into memory. This in turn requires to reason more about the log layout as we cannot afford millions of small random gets and puts on SSD like in memory. Another aspect is related to the update patterns, depending on the applications. If an application updates a few objects many times with the described on-demand reorganization, the log is increasingly populated with many outdated log entries. This slows down the recovery process as all log entry headers need to be analyzed during recovery. We plan to address these problems by an incremental and periodical reorganization scheme.
In [RO92] the authors have shown that the reorganization overhead can be reduced by dividing the log into segments and monitoring write frequency of each segment. The authors then propose to prefer reorganizing segments that are updated less frequently as these segments tend to stay longer organized. This in turn avoids repeated moves of unchanged log entries and leads to a categorization of log entries depending on their update frequency (cold and hot segments). The selection of segments is based on the cost-benefit policy (equation 1) in which ”u” the utilization of the segment and ”age” the age of the youngest

log entry in that segment is. This means the desired segment contains a few, for a long unchanged entries, hence much space can be regained and the resulting new less-filled segment includes data that was changed less frequently in the past.

$$\frac{\text{benefit}}{\text{cost}} = \frac{\text{free space generated} * \text{age of data}}{\text{cost}} = \frac{(1-u) * \text{age}}{1+u} \tag{1}$$

While this strategy enables the reorganization of logs with low additional write costs there are also some shortcomings. First of all this approach requires maintaining some monitoring data in memory per log entry to determine the segment with best cost-benefit coefficient and to decide which log entries are alive. Furthermore, by reorganizing only a subset of log entries at a given time, the system can no longer determine the newest version of an log entry by its position in log (newer versions are nearer to the end of the log). Thus we need to add a version number to the log entry headers. Considering the sheer amount of chunks we expect, this might be an issue. On the other hand by giving up the strict ordering of log entries, techniques like threading and hole-plugging [MRC+97] or slack space recycling [OKC+10] are applicable.

Another interesting approach is to distribute log entries into different logs based on write statistics. This means chunks that are changed frequently are bundled in one log and those that are changed seldom in another one (more than two partitions conceivable). This creates hot and cold zones like with the cost-benefit policy but with shifting the complexity from reorganization to storing of backups. The advantage of this approach is the rather simple reorganization, because the segment selection is less important. However both variants are adaptable which will be an important aspect in finding a space and time efficient reorganization scheme for logging billions of small chunks.

6 Related Work

Logging has been used in different research areas but because of the limited space in this paper we discuss only the most relevant related work.

As DXRAM is inspired by RAMCloud [OAE+10] it is sharing many ideas but there are also important differences, including the logging approach. First of all, RAMCloud is providing a table-based data model and is not aiming at supporting billions of small data objects like DXRAM does. The in-memory layout is a 1:1 copy of the remote log which fits well for a table-based model but not as good for many small objects. The reorganization of the disk logs is controlled by the storage node reorganizing segments in memory and writing the compacted results to new segments on potentially new backup nodes. While the reorganization can run in memory this introduces a heavy network traffic. Therefore, we decided to allow backup nodes to do a reorganization for their secondary logs independent of other backup logs and the storage node. Furthermore RAMCloud does not have the two stage logging approach with the sorting of log entries per node as we have. Finally, it is worth to note that RAMCloud has successfully shown that it is possible to recover the state of node with 35 GB of in-memory data in around 1.6 seconds [ORS+11b]. This is

a lower bound as the authors have used 1,000 backup nodes for recovering one node in parallel and used an Infiniband high-speed network.

The idea behind log-structured file systems like SpriteFS [RO92] is to minimize the movements of the actuator arm in traditional HDDs, because moving the actuator arm to the right position and waiting for the rotation until the desired block is accessible can take numerous milliseconds. As a consequence SpriteFS tried to write sequentially whenever possible, which in turn means that block updates are appended with higher version instead of replacing blocks in place. Read accesses are mostly served from the block cache avoiding heavy random reads on the HDD. With time the log becomes overfilled with old versions and lacks free space to store new objects or updates. SpriteFS addressed this by dividing the log into segments and execute a cleaner on segments periodically. During the cleaning, old versions are deleted and the segment is defragmented. After defragmentation the space of the old segment is freed and the remaining objects are appended to the log. In addition the cleaner does not choose segments with the same probability but uses a cost benefit policy to maximize the amount of space that is regained with every processed segment.

SFS is a more recent work transferring the advantages of a log-structured file system from HDDs to SSDs [MKC+12]. For better performance the segment size fits the block size of the SSDs. Another interesting aspect is that the hot and cold zones are generated proactively with statistics on block level to increase the efficiency of the defragmentation. It is future work for DXRAM to refine the cost benefit policy and to optimize it for billions of small data objects stored on SSDs.

Logging has also a long tradition in database systems, where typically a write-ahead log (WAL) is used. However, these logs are totally different from the one proposed in this paper. WAL logs do not store the data per se but the transaction's meta-data to apply transactions on backup databases or to repeat failed transactions. The data objects are stored on persistent memory anyway. If the log is overfilled old log entries are flushed to the database and will be overwritten. So, there is no need for a log cleaner.

Finally, there have been proposed many SSD-based key value stores, which however typically use SSDs as caches for HDDs in order to speed up data accesses. One example is Flashstore [DSL10], which uses a cuckoo-based hash-table with compact key signatures as index structure in RAM to improve access times for reading objects from SSD. While the related insights are interesting for meta-data management in DXRAM, they do not affect the log architecture.

7 Conclusion

DXRAM is a distributed in-memory system that is designed to manage billions of small data objects. By keeping all data always in memory it is providing low-latency data access to all data, while at the same time relieving programmers from keeping caches and secondary storage synchronized. The latter requires an efficient asynchronous logging of in-memory data on remote flash disks for providing fast recovery from node failures and

avoiding data loss.

We have presented a novel two stage logging approach sorting log entries per node allowing fast recovery of failed nodes. The latter is also supported by spreading the state of one node on potentially many backup node logs, which allows a parallel recovery from many nodes. As known from the systems literature, sequential logging is fast but requires a cleaner to remove outdated and deleted data and compact the log to free space. We believe that the cost-benefit approach like proposed for log-structured files systems is a good base for a more optimized custom solution for an in-memory based storage.

Future work includes the design of cleaner policies as well as evaluations with different application access patterns (including recovery times depending on different configurations). Currently, we are adopting the BGbenchmark [BG13], which allows to run actions of social network applications on top of different storage technologies (e.g. MongoDB and SQL-X with memcached). We plan to use this benchmark for comparing DXRAM performance with key-value and distributed in-memory caches (e.g. Gemfire and Hazelcast) as well as the recovery times for single and multiple node failures.

References

[AXF+12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 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, pages 53–64, New York, NY, USA, 2012. ACM.

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[BG13] Sumita Barahmand and Shahram Ghandeharizadeh. BG: A Benchmark to Evaluate Interactive Social Networking Actions. In *CIDR*, 2013.

[CKZ09] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[DSL10] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3(1-2):1414–1425, September 2010.

[KS13] F. Klein and M. Schoettner. Dxram: A persistent in-memory storage for billions of small objects. In *Proceedings of the 14th International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT 13, 2013.

[MKC+12] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the*

10th USENIX Conference on File and Storage Technologies, FAST'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-structured File Systems with Adaptive Methods. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 238–251, New York, NY, USA, 1997. ACM.

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[OAE+10] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev.*, 43(4):92–105, January 2010.

[OKC+10] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Optimizations of LFS with Slack Space Recycling and Lazy Indirect Block Update. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, SYSTOR '10, pages 2:1–2:9, New York, NY, USA, 2010. ACM.

[ORS+11a] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[ORS+11b] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[RO92] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.

[SWL13] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[SWW+12] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.*, 5(9):788–799, May 2012.

Acronyms

DRAM Dynamic Random Access Memory

RAM Random Access Memory

CLI Command Line Interface

RMI Remote Method Invocation

RPC Remote Procedure Call

MPI Message Passing Interface

DSM Distributed Shared Memory

RDMA Remote Direct Memory Access

IPC Inter-Process Communication

HTTP Hypertext Transfer Protocol

REST Representational State Transfer

NVMe Non-Volatile Memory Express

PCI-e Peripheral Component Interconnect Express

HDD Hard Disk Drive

SSD Solid State Disk

JVM Java Virtual Machine

JNI Java Native Interface

SLA Service Layer Agreement

TCP Transmission Control Protocol

IP Internet Protocol

CRC Cyclic Redundany Check

RTT Round Trip Time

FC Flow Control

OOB Out-Of-Band Data

HCA Host Channel Adapter

CAS Compare-And-Set

RDD Resilient Distributed Dataset

RAF Random Access File

YCSB Yahoo! Cloud Serving Benchmark

ORB Outgoing Ring Buffer

IBQ Incoming Buffer Queue

CUB Catch-Up Buffer

MCC Message Creation Coordinator

BFS Breadth-First Search

CID Chunk ID

LID Local ID

ZID Zone ID

Bibliography

[1] *A Java serialization/deserialization library to convert Java Objects into JSON and back*. `https://github.com/google/gson`. Accessed: 2018-06-05 (Page: 108).

[2] *Apache Mahout*. `https://mahout.apache.org/`. Accessed: 2018-06-21 (Page: 8).

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. 2015, pp. 1383–1394 (Page: 9).

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "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. 2012, pp. 53–64 (Page: 2).

[5] Hagit Attiya and Jennifer L. Welch. "Sequential Consistency versus Linearizability." In: 12 (May 1994), pp. 91–122 (Page: 11).

[6] Sumita Barahmand and Shahram Ghandeharizadeh. "BG: A Benchmark to Evaluate Interactive Social Networking Actions". In: *CIDR*. 2013 (Pages: 16, 17).

[7] Kevin Beineke. "Scalable Distributed Metadata Management of Many Small Objects". Master's Thesis. Universitaetsstrasse 1, 40225 Duesseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Duesseldorf, Dec. 2013 (Pages: 14, 112).

[8] Kevin Beineke, Florian Klein, Stefan Nothaas, and Michael Schoettner. *DXRAM Project on Github*. `https://github.com/hhu-bsinfo/dxram`. Accessed: 2018-06-02 (Page: 14).

[9] Kevin Beineke, Florian Klein, Stefan Nothaas, and Michael Schoettner. *DXRAM Project Website*. `https://dxram.io`. Accessed: 2018-06-04 (Page: 14).

[10] Kevin Beineke, Florian Klein, and Michael Schoettner. "Asynchronous logging and fast recovery for a large-scale distributed in-memory storage". In: *Informatik 2014*. Ed. by E. Plödereder, L. Grunske, E. Schneider, and D. Ull. Bonn: Gesellschaft für Informatik e.V., 2014, pp. 1797–1810 (Pages: 56, 57, 115).

[11] Kevin Beineke, Stefan Nothaas, Florian Klein, and Michael Schoettner. *DXNet Project on Github*. `https://github.com/hhu-bsinfo/dxnet`. Accessed: 2018-06-02 (Page: 27).

[12] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "DXRAM's Fault-Tolerance Mechanisms Meet High Speed I/O Devices". In: *ArXiv e-prints* (July 2018). arXiv: `1807.03562 [cs.DC]` (Pages: 13, 20, 24–26, 32, 80).

[13] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Efficient Messaging for Java Applications Running in Data Centers". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2018, pp. 589–598 (Pages: 13, 27, 28, 31, 44, 80).

[14] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Fast Parallel Recovery of Many Small In-Memory Objects". In: *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2017, pp. 248–257 (Pages: 6, 7, 11, 13, 21, 26, 27, 32, 45, 68, 80).

[15] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "High Throughput Log-Based Replication for Many Small In-Memory Objects". In: *IEEE 22nd International Conference on Parallel and Distributed Systems*. Dec. 2016, pp. 535–544 (Pages: 6, 13, 17, 23, 32, 45, 56, 68, 69, 80).

[16] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "High Throughput Log-Based Replication for Many Small In-Memory Objects". In: *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2016, pp. 160–161 (Page: 56).

[17] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Parallelized Recovery of Hundreds of Millions Small Data Objects". In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2017, pp. 621–622 (Page: 68).

[18] Kevin Beineke, Stefan Nothaas, and Michael Schoettner. "Scalable Messaging for Java-based Cloud Applications". In: *ICNS 2018, The Fourteenth International Conference on Network and Services* 14 (May 2018), pp. 32–41 (Pages: 27–30, 44, 113).

[19] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. "Apache Hadoop Goes Realtime at Facebook". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. 2011, pp. 1071–1080 (Page: 9).

[20] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX ATC'13. 2013, pp. 49–60 (Pages: 2, 7, 9).

[21] Josiah L. Carlson. *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013 (Pages: 3, 5, 16).

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *ACM Trans. Comput. Syst.* 26 (June 2008), 4:1–4:26 (Pages: 1, 26).

[23] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2016, pp. 1789–1792 (Page: 8).

[24] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. "Copysets: Reducing the Frequency of Data Loss in Cloud Storage". In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013 (Pages: 10, 80, 81).

[25] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. "Benchmarking cloud serving systems with YCSB". In: *Proc. of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154 (Page: 16).

[26] HPC Advisory Council. *Interconnect Analysis: 10GigE and InfiniBand in High Performance Computing*. White Paper. 350 Oakmead Pkwy, Sunnyvale, CA 94085, 2009 (Page: 8).

[27] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Commun. ACM* 51 (Jan. 2008), pp. 107–113 (Pages: 7, 8).

[28] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. "Dynamo: amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41 (Oct. 2007), pp. 205–220 (Page: 1).

[29] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker, and David A. Wood. "Implementation Techniques for Main Memory Database Systems". In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. SIGMOD '84. 1984, pp. 1–8 (Page: 1).

[30] Jens Dittrich and Jorge-Arnulfo Quiané-Ruiz. "Efficient Big Data Processing in Hadoop MapReduce". In: *Proc. VLDB Endow.* 5 (Aug. 2012), pp. 2014–2015 (Page: 8).

[31] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. "FaRM: Fast Remote Memory". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Apr. 2014, pp. 401–414 (Pages: 2, 6, 12).

[32] Margaret H. Eich. "Mars: The Design of a Main Memory Database Machine". In: *Database Machines and Knowledge Base Machines*. Ed. by Masaru Kitsuregawa and Hidehiko Tanaka. Boston, MA: Springer US, 1988, pp. 325–338 (Page: 1).

[33] Martin S Engler, Mohammed El-Kebir, Jelmer Mulder, Alan E Mark, Daan P Geerke, and Gunnar W Klau. "Enumerating common molecular substructures". In: *PeerJ Preprints* 5 (Sept. 2017), e3250v1 (Pages: 2, 16).

[34] Marc Ewert. "A thread-pool-based network interface for a distributed in-memory storage". Master's Thesis. Universitaetsstrasse 1, 40225 Duesseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Duesseldorf, Feb. 2015 (Page: 32).

[35] *Fast Restart*. `https://www.aerospike.com/docs/operations/manage/aerospike/fast_start/index.html`. Accessed: 2018-07-03 (Page: 5).

[36] T. Fields and B. Cotton. *Social Game Design: Monetization Methods and Mechanics*. Morgan Kauffman Publishers, 2011 (Page: 2).

[37] Brad Fitzpatrick. "Distributed caching with memcached". In: *Linux journal* 2004 (2004) (Pages: 1, 5, 9).

[38] A. Fuad, A. Erwin, and H. P. Ipung. "Processing performance on Apache Pig, Apache Hive and MySQL cluster". In: *Proceedings of International Conference on Information, Communication Technology and System (ICTS) 2014*. Sept. 2014, pp. 297–302 (Page: 9).

[39] Christian Gesse. *Optimizing Access to SSD for Backups in In-Memory Systems*. Universitaetsstrasse 1, 40225 Duesseldorf, Germany, Nov. 2016 (Page: 81).

[40] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. "Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications". In: *SIGCOMM Comput. Commun. Rev.* 41 (Aug. 2011), pp. 350–361 (Page: 4).

[41] *GitHub - antirez/redis: Redis is an in-memory database that persists on disk. The data model is key-value, but many different kind of values are supported: Strings, Lists, Sets, Sorted Sets, Hashes, HyperLogLogs, Bitmaps*. `https://github.com/antirez/redis`. Accessed: 2018-07-02 (Page: 8).

[42] *GitHub - Microsoft/GraphEngine: Microsoft Graph Engine*. `https://github.com/Microsoft/GraphEngine`. Accessed: 2018-07-02 (Page: 8).

[43] Trey Grainger and Timothy Potter. *Solr in Action*. Greenwich, CT, USA: Manning Publications Co., Mar. 2014 (Page: 8).

[44] Antonio Gulli and Alessio Signorini. "The indexable web is more than 11.5 billion pages". In: *Special interest tracks and posters of the 14th international conference on World Wide Web*. 2005, pp. 902–903 (Page: 3).

[45] *Hazelcast Homepage*. `http://www.hazelcast.com` (Page: 8).

[46] Joel Hruska. *Intel, Micron reveal Xpoint, a new memory architecture that could outclass DDR4 and NAND*. `https://www.extremetech.com/extreme/211087-intel-micron-reveal-xpoint-a-new-memory-architecture-that-claims-to-outclass-both-ddr4-and-nand`. Accessed: 2018-06-20. 2015 (Page: 1).

[47] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. 2010, pp. 11–11 (Pages: 9, 111).

[48] Mohammad Islam, Angelo K. Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. "Oozie: Towards a Scalable Workflow Management System for Hadoop". In: *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*. SWEET '12. 2012, 4:1–4:10 (Page: 9).

[49] *Jline 2.x*. `https://github.com/jline/jline2`. Accessed: 2018-06-05 (Page: 108).

[50] E. Jovanov. "Wireless Technology and System Integration in Body Area Networks for m-Health Applications". In: *2005 IEEE Engineering in Medicine and Biology 27th Annual Conference*. Jan. 2005, pp. 7158–7160 (Pages: 1, 2).

[51] Yunus Kaplan. "A SSD-aware logging service for a distributed in-memory storage". Master's Thesis. Universitaetsstrasse 1, 40225 Duesseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Duesseldorf, Mar. 2014 (Page: 57).

[52] Florian Klein. "Metadata-Management in a distributed In-Memory Storage". PhD thesis. Universitaetsstrasse 1, 40225 Duesseldorf, Germany: Institute for Computer Science, Heinrich-Heine University Duesseldorf, Nov. 2015 (Pages: 14, 21).

[53] Florian Klein, Kevin Beineke, and Michael Schoettner. "Distributed Range-Based Meta-Data Management for an In-Memory Storage". In: *LNCS Europar Workshop Proceedings, 4th Big Workshop on Big Data Managements in Clouds*. Sept. 2015 (Pages: 9, 17).

[54] Florian Klein, Kevin Beineke, and Michael Schoettner. "Memory Management for Billions of Small Objects in a Distributed In-Memory Storage". In: *IEEE Cluster 2014*. Sept. 2014 (Page: 6).

[55] Florian Klein and Michael Schoettner. "Dxram: A persistent in-memory storage for billions of small objects". In: *Proceedings of the 14th International Conference on Parallel and Distributed Computing, Applications and Technologies*. PDCAT 13. 2013 (Page: 9).

[56] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. "Impala: A Modern, Open-Source SQL Engine for Hadoop". In: *CIDR*. 2015 (Page: 9).

[57] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. "What is Twitter, a Social Network or a News Media?" In: *Proceedings of the 19th International Conference on World Wide Web*. WWW '10. 2010, pp. 591–600 (Pages: 1, 2).

[58] Avinash Lakshman and Prashant Malik. "Cassandra: a decentralized structured storage system". In: *SIGOPS Oper. Syst. Rev.* 44 (Apr. 2010), pp. 35–40 (Pages: 9, 17).

[59] Leslie Lamport. "Paxos Made Simple". In: (Dec. 2001), pp. 51–58 (Page: 6).

[60] Charles E. Leiserson and Tao B. Schardl. "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)". In: *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*. SPAA '10. 2010, pp. 303–314 (Page: 19).

[61] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: SOCC '14. 2014 (Pages: 2, 4, 7).

[62] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 429–444 (Page: 8).

[63] P.D. Londhe, S.S. Kumbhar, R.S. Sul, and A.J. Khadse. "Processing big data using hadoop framework". In: *Proceedings of 4th SARC-IRF International Conference*. Apr. 2014, pp. 72–75 (Page: 9).

[64] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud". In: *Proc. VLDB Endow.* 5 (Apr. 2012), pp. 716–727 (Pages: 3, 16).

[65] Hongjun Lu, Yuet Yeung Ng, and Zengping Tian. "T-tree or B-tree: main memory database index structure revisited". In: *Database Conference, 2000. ADC 2000. Proceedings. 11th Australasian*. 2000, pp. 65–73 (Page: 20).

[66] Yifeng Luo, Siqiang Luo, Jihong Guan, and Shuigeng Zhou. "A RAMCloud Storage System Based on HDFS: Architecture, Implementation and Evaluation". In: *J. Syst. Softw.* 86 (Mar. 2013), pp. 744–750 (Page: 9).

[67] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. 2010, pp. 135–146 (Page: 7).

[68] Dahlia Malkhi and Michael Reiter. "Byzantine quorum systems". In: *Distributed Computing* 11 (Oct. 1998), pp. 203–213 (Page: 4).

[69] W. Glynn Mangold and David J. Faulds. "Social media: The new hybrid element of the promotion mix". In: *Business Horizons* 52 (2009), pp. 357–365 (Pages: 1, 2).

[70] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. "Use at Your Own Risk: The Java Unsafe API in the Wild". In: *SIGPLAN Not.* 50 (Oct. 2015), pp. 695–710 (Page: 18).

[71] Michael McCandless, Erik Hatcher, and Otis Gospodnetic. *Lucene in Action, Second Edition: Covers Apache Lucene 3.0*. Greenwich, CT, USA: Manning Publications Co., 2010 (Page: 8).

[72] Sneha Mehta and Viral Mehta. "Hadoop Ecosystem: An Introduction". In: *International Journal of Science and Research (IJSR)*. Vol. 5. June 2016 (Page: 8).

[73] *Memcached SPOF Mystery*. `https://blog.twitter.com/engineering/en_us/a/2010/memcached-spof-mystery.html`. Accessed: 2018-06-25 (Page: 9).

[74] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17 (2016), pp. 1–7 (Page: 8).

[75] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. "Graph Structure in the Web — Revisited: A Trick of the Heavy Tail". In: *Proceedings of the 23rd International Conference on World Wide Web*. WWW '14 Companion. 2014, pp. 427–432 (Pages: 1, 3, 4).

[76] 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)*. 2013, pp. 103–114 (Page: 12).

[77] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. "Introducing the graph 500". In: (2010) (Page: 16).

[78] Maryam M. Najafabadi, Flavio Villanustre, Taghi M. Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. "Deep learning applications and challenges in big data analytics". In: *Journal of Big Data* 2 (Feb. 2015) (Page: 7).

[79] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. *Grappa: A latency-tolerant runtime for large-scale irregular applications*. Technical Report. 2014 (Page: 16).

[80] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski and Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 385–398 (Pages: 1, 2, 9).

[81] Stefan Nothaas, Kevin Beineke, and Michael Schöttner. "Distributed Multithreaded Breadth-first Search on Large Graphs Using DXGraph". In: *Proceedings of the First International Workshop on High Performance Graph Data Management and Processing*. HPGDMP '16. 2016, pp. 1–8 (Pages: 16, 18, 19).

[82] S. Oikawa and R. Rajkumar. May 1998 (Page: 109).

[83] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. 2014, pp. 305–319 (Page: 111).

[84] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. 2011, pp. 29–41 (Pages: 2, 3, 6, 16, 26).

[85] *Open source memory-centric distributed database, caching and processing platform - Apache Ignite*. `https://ignite.apache.org/index.html`. Accessed: 2018-06-21 (Page: 9).

[86] Mohd Fauzi Othman and Khairunnisa Shazali. "Wireless Sensor Network Applications: A Study in Environment Monitoring System". In: *Procedia Engineering* 41 (2012), pp. 1204–1210 (Pages: 1, 2).

[87] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. "The case for RAMClouds: scalable high-performance storage entirely in DRAM". In: *SIGOPS Oper. Syst. Rev.* 43 (Jan. 2010), pp. 92–105 (Pages: 1, 2, 6, 8, 12).

[88] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33 (Aug. 2015), 7:1–7:55 (Page: 2).

[89] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. *The PageRank Citation Ranking: Bringing Order to the Web*. Technical Report 1999-66. Previous number = SIDL-WP-1999-0120. Nov. 1999 (Pages: 1, 2).

[90] Eelco Plugge, Tim Hawkins, and Peter Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. 1st. Berkely, CA, USA: Apress, 2010 (Pages: 3, 17).

[91] *PoweredBy - Hadoop Wiki*. `https://wiki.apache.org/hadoop/PoweredBy`. Accessed: 2018-06-21 (Page: 8).

[92] Dan Pritchett. "BASE: An Acid Alternative". In: *Queue* 6 (May 2008), pp. 48–55 (Page: 3).

[93] M. J. Rashti and A. Afsahi. "10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G". In: *2007 IEEE International Parallel and Distributed Processing Symposium*. Mar. 2007, pp. 1–8 (Page: 12).

[94] *reddit's May 2010 "State of the Servers" report*. `https://redditblog.com/2010/05/11/reddits-may-2010-state-of-the-servers-report/`. Accessed: 2018-06-25 (Page: 9).

[95] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. "Handling Churn in a DHT". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '04. 2004, pp. 10–10 (Page: 4).

[96] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. "Pickling State in the javaTM System". In: *Proc. of the 2nd Conf. on USENIX Conf. on Object-Oriented Technologies*. 1996, pp. 19–19 (Page: 8).

[97] Mendel Rosenblum and John K. Ousterhout. "The Design and Implementation of a Log-structured File System". In: *ACM Trans. Comput. Syst.* 10 (Feb. 1992), pp. 26–52 (Page: 81).

[98] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-structured Memory for DRAM-based Storage". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. 2014, pp. 1–16 (Pages: 6, 10, 26).

[99] M. Samovsky and T. Kacur. "Cloud-based classification of text documents using the Gridgain platform". In: *2012 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*. May 2012, pp. 241–245 (Page: 3).

[100] *SAP HANA Backup and Recovery (Overview)*. `https://www.sap.com/documents/2015/03/c86ff654-5a7c-0010-82c7-eda71af511fa.html`. Accessed: 2018-07-10 (Pages: 2, 5).

[101] *SAP HANA Customer Reviews*. `https://www.sap.com/products/hana/customer-reviews.html`. Accessed: 2018-06-26 (Page: 5).

[102] Pratik Satapathy, Jash Dave, Priyanka Naik, and Mythili Vutukuru. "Performance Comparison of State Synchronization Techniques in a Distributed LTE EPC". In: *IEEE Conf. on Network Function Virtualization and Software Defined Networks*. 2017 (Page: 2).

[103] *Seattle Conference on Scalability: YouTube Scalability*. `https://www.youtube.com/watch?v=ZW5_eEKEC28`. Accessed: 2018-06-25 (Page: 9).

[104] Bin Shao, Haixun Wang, and Yatao Li. "Trinity: A Distributed Graph Engine on a Memory Cloud". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. 2013, pp. 505–516 (Page: 6).

[105] E. Shurman and J. Brutlag. *Performance related changes and their user impact*. `http://oreil.ly/fTmYwz`. 2009 (Pages: 1, 2).

[106] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. "The Hadoop Distributed File System". In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. May 2010, pp. 1–10 (Page: 9).

[107] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. "Efficient transaction processing in SAP HANA database: the end of a column store myth". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. 2012, pp. 731–742 (Pages: 3, 5).

[108] Dilpreet Singh and Chandan K. Reddy. "A survey on platforms for big data analytics". In: *Journal of Big Data* 2 (Oct. 2014), p. 8 (Page: 7).

[109] Sergej Sizov and Mikle Bahn. "Disciplinary assessment of scientific content by higher-order citation mining". In: *Proceedings of the Association for Information Science and Technology* 54 (Oct. 2017), pp. 383–393 (Page: 16).

[110] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. "Aerospike: Architecture of a Real-time Operational DBMS". In: *Proc. VLDB Endow.* 9 (Sept. 2016), pp. 1389–1400 (Page: 8).

[111] *SSD Ranking: The Fastest Solid State Drives 2018*. `http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/#m2`. Accessed: 2018-07-03 (Page: 4).

[112] *Statistics | TOP500 Supercomputer Sites*. `https://www.top500.org/statistics/`. Accessed: 2018-03-17 (Pages: 1, 12, 45).

[113] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications". In: *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '01. 2001, pp. 149–160 (Page: 20).

[114] Ryan Stutsman. "Durability and crash recovery in distributed inmemory storage systems". dissertation. Stanford University, The Department of Computer Science, Stanford, CA, USA, 2013 (Page: 6).

[115] Wenhui Tang, Yutong Lu, Nong Xiao, Fang Liu, and Zhiguang Chen. "Accelerating Redis with RDMA Over InfiniBand". In: *Data Mining and Big Data*. Ed. by Ying Tan, Hideyuki Takagi, and Yuhui Shi. Cham: Springer International Publishing, 2017, pp. 472–483 (Page: 8).

[116] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. "The Anatomy of the Facebook Social Graph". In: *CoRR* abs/1111.4503 (2011) (Pages: 1, 2, 4, 7).

[117] Brian Bulkowski V. Srinivasan. "Citrusleaf: A Real-Time NoSQL DB which Preserves ACID". In: (Aug. 2011) (Pages: 3, 5, 6).

[118] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. 2013, 5:1–5:16 (Page: 9).

[119] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda. "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems". In: *2012 IEEE 20th Annual Symposium on High-Performance Interconnects (HOTI)*. Aug. 2012, pp. 48–55 (Page: 12).

[120] Mehul Nalin Vora. "Hadoop-HBase for large-scale data". In: *Proceedings of 2011 International Conference on Computer Science and Network Technology*. Vol. 1. Dec. 2011, pp. 601–605 (Page: 9).

[121] Sameer Wadkar and Madhu Siddalingaiah. "Apache Ambari". In: *Pro Apache Hadoop*. Berkeley, CA: Apress, 2014 (Page: 9).

[122] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. "Building a Replicated Logging System with Apache Kafka". In: *Proc. VLDB Endow.* 8 (Aug. 2015), pp. 1654–1655 (Page: 8).

[123] S. Wang. *Graph Analytics and Machine Learning*. `https://de.slideshare.net/stanleywanguni/graph-analytic-and-machine-learning`. 2016 (Page: 2).

[124] *Who's using Redis*. `https://redis.io/topics/whos-using-redis`. Accessed: 2018-06-25 (Page: 5).

[125] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. "GraphX: A Resilient Distributed Graph System on Spark". In: *First International Workshop on Graph Data Management Experiences and Systems*. GRADES '13. 2013, 2:1–2:6 (Pages: 3, 8).

[126] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. "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)*. 2012, pp. 15–28 (Page: 9).

[127] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. 2010, pp. 10–10 (Pages: 7–9).

[128] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. "In-Memory Big Data Management and Processing: A Survey". In: *IEEE Transactions on Knowledge and Data Engineering* 27.7 (July 2015), pp. 1920–1948 (Pages: 1–3, 5, 7).

[129] Weinan Zhang, Shuai Yuan, and Jun Wang. "Optimal Real-time Bidding for Display Advertising". In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '14. 2014, pp. 1077–1086 (Page: 2).

[130] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. "Tapestry: A Fault-tolerant Wide-area Application Infrastructure". In: *SIGCOMM Comput. Commun. Rev.* 32 (Jan. 2002), pp. 81–81 (Page: 4).

[131] Yu Zheng, Quannan Li, Yukun Chen, Xing Xie, and Wei-Ying Ma. "Understanding Mobility Based on GPS Data". In: *Proceedings of the 10th International Conference on Ubiquitous Computing*. UbiComp '08. 2008, pp. 312–321 (Page: 1).

Kevin Beineke

Personal Details

Name:	Kevin Josef Beineke
Birth:	November 25th 1987 in Düsseldorf, Germany
Nationality:	German

Languages

German:	native
English:	written and spoken
Spanish:	school knowledge
Latin:	qualification in Latin

Academic and professional experience

Since 01/2014	Doctoral studies at the Heinrich-Heine University Düsseldorf
12/2013	Degree: Master of Science (Final grade: 1.1)
	Master thesis: „Scalable Distributed Metadata Management for Many Small Objects“ Grade: 1.0
10/2011 - 12/2013	Studies of Computer Science (Master) Heinrich-Heine University Düsseldorf Focus: distributed and parallel systems, computer networks, peer-to-peer networks, neural networks
10/2011	Degree: Bachelor of Science (Final grade: 1.5)
	Bachelor thesis: „Adaptive Replication in a Distributed In-Memory System“ Grade: 1.0
10/2008 - 9/2011	Studies of Computer Science (Bachelor) Heinrich-Heine University Düsseldorf Focus: operating systems, computer networks Secondary subject: biology
1/2008 - 9/2008	Basic military service Award: Bestpreis
6/2007	Abitur (Final grade: 2.3) Comenius Gymnasium, Düsseldorf, Germany

Practical experiences

04/2010 - 07/2010	Heinrich-Heine University Düsseldorf Department: Computer Networks Student assistant Tutor for "Computer Science II" Participated in a course regarding the principles of teaching
10/2010 - 12/2013	Heinrich-Heine University Düsseldorf Department: Operating Systems Student assistant Writing an application for OSS (Object Sharing Service) Developing, porting and extending benchmarks for ECRAM Developing and extending the function set of ECRAM Adding fault-tolerance aspects to ECRAM Evaluating ECRAM: scalability comparison between cluster and HPC (HP DL980; Future SOC Lab of the Hasso-Plattner institute) Developing an adaptive replication scheme for ECRAM Developing the global metadata management of DXRAM Porting a social media benchmark for DXRAM (new data model)
01/2014 - 06/2015	Heinrich-Heine University Düsseldorf Centre for Information and Media Technology Research assistant Deployment and maintenance of databases (IBM Informix, PostgreSQL, MySQL and Microsoft SQL Server) License demand calculation Porting online database from IBM Informix to PostgreSQL
Since 01/2014	Heinrich-Heine University Düsseldorf Department: Operating Systems Research assistant Distributed and parallel infrastructures for big data processing

Research projects

2010	OSS (Object Sharing Service) Distributed shared memory system with transactional consistency Part of the European project XtreemOS Tasks: benchmarking and evaluation
2011 - 2013	ECRAM (Elastic Cooperative Random Access Memory) Distributed shared memory system with transactional consistency Based on OSS, extended by a MapReduce framework and file system Tasks: adaptive replication, fault-tolerance, benchmarking
Since 2013	DXRAM (Distributed eXtreme Memory) Distributed in-memory key-value store Optimized for low-latency access and resource efficiency
Since 2017	DXNet Network messaging system for Java applications Standalone; used by DXRAM

Teaching

Involvement in the following courses at the Heinrich-Heine University Düsseldorf:

Exercises/Project	Distributed and Parallel Systems SS 2014
Exercises	Computer Science I WS 2014/2015
Lab exercises	Operating Systems WS 2015/2016
Exercises	Fundamentals Distributed Systems WS 2015/2016
Seminar	Architecture Distributed Systems SS 2016
Lab exercises	Distributed and Parallel Systems SS 2016
Exercises	Computer Science I WS 2016/2017
Seminar	Virtual & Augmented Reality WS 2016/2017
Exercises/Project	Operating Systems SS 2017
Exercises	Fundamentals Distributed Systems SS 2017
Seminar	Architecture Distributed Systems WS 2017/2018
Lab exercises	Distributed and Parallel Programming WS 2017/2018
Lab exercises	Operating Systems and System Programming 2018

Personal Publications

Reviewed conference papers

Kim-Thomas Rehmann, Kevin Beineke and Michael Schöttner. "Smart Replication for In-memory Computations". In: *Proc. of the 18th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2012. Acceptance Rate: 29.6%.

Florian Klein, Kevin Beineke and Michael Schöttner. "Memory Management for Billions of Small Objects in a Distributed In-Memory Storage". In: *Proc. of the 2014 IEEE International Conference on Cluster Computing (CLUSTER)*. 2014. Acceptance Rate: 23.8%.

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "High Throughput Log-based Replication for Many Small In-Memory Objects". In: *Proc. of the 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2016. Acceptance Rate: 29.9%.

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Fast Parallel Recovery of Many Small In-memory Objects". In: *Proc. of the 23rd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. 2017. Acceptance Rate: 32.8%.

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Scalable Messaging for Java-based Cloud Applications". In: *ICNS 2018, The Fourteenth International Conference on Networking and Services*. 2018. Acceptance Rate: 29%.

Reviewed workshop papers

Kevin Beineke, Florian Klein and Michael Schöttner. "Asynchronous Logging and Fast Recovery for a Large-Scale Distributed In-Memory Storage". In: *INFORMATIK 2014 proceedings, BigSys14 workshop*. 2014.

Florian Klein, Kevin Beineke and Michael Schöttner. "Distributed Range-Based Meta-Data Management for an In-Memory Storage". In: *LNCS Europar Workshop Proceedings, 4th Big Workshop on Big Data Managements in Clouds*. 2015.

Stefan Nothaas, Kevin Beineke and Michael Schöttner. "Distributed Multithreaded Breadth-First Search on Large Graphs using DXGraph". In: *Proc. of the 1st High Performance Graph Data Management and Processing workshop (HPGDMP)*. 2016.

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Efficient Messaging for Java Applications running in Data Centers". In: *Cluster, Cloud and Grid Computing (CC-GRID), 2018 18th IEEE/ACM International Symposium on, Workshop AHPAMA*. 2018.

Reviewed two-page short papers for poster presentation

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "High Throughput Log-based Replication for Many Small In-Memory Objects". In: *Proc. of the 2016 IEEE International Conference on Cluster Computing (CLUSTER), Poster*. 2016.

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "Parallelized Recovery of Hundreds of Millions of Small Data Objects". In: *Proc. of the 2017 IEEE International Conference on Cluster Computing (CLUSTER), Poster*. 2017.

Reports

Kevin Beineke, Stefan Nothaas and Michael Schöttner. "DXRAM's Fault-Tolerance Mechanisms Meet High Speed I/O Devices". In: ArXiv e-prints (July 2018). arXiv:1807.03562 [cs.DC].

Eidesstattliche Erklärung
laut §5 der Promotionsordnung vom 06.12.2013

Ich versichere an Eides Statt, dass die Dissertation von mir selbständig und ohne unzulässige fremde Hilfe unter Beachtung der „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Heinrich-Heine-Universität Düsseldorf“ erstellt worden ist.

Ort, Datum	Kevin Beineke