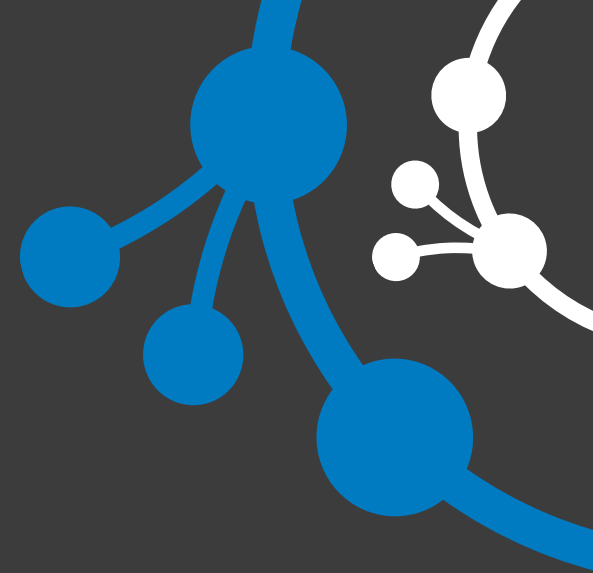


DXRAM: A Distributed In-Memory Key-Value Storage Optimized for Small Data Objects



Department of Computer Science
Heinrich-Heine-University Düsseldorf, Germany

Last update: June 06, 2019



Contributors

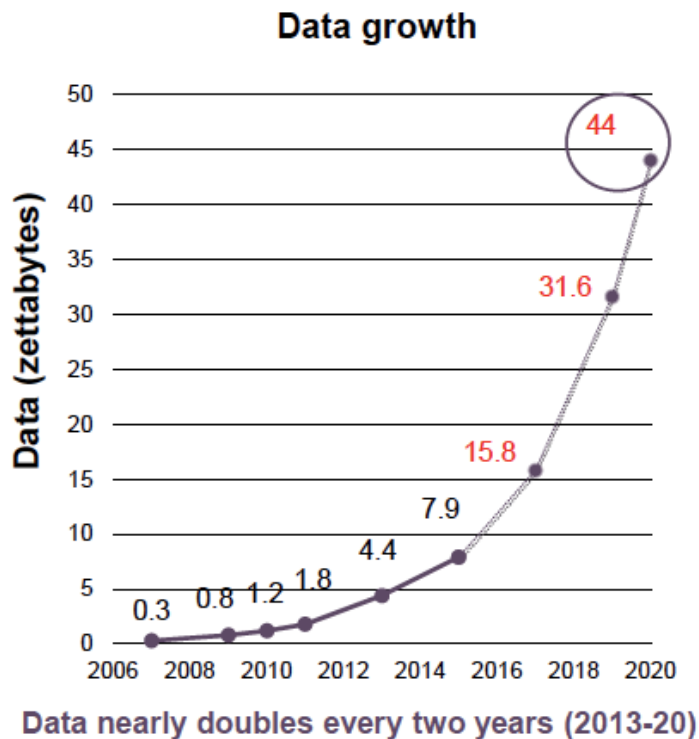
- **Dr. Florian Klein** (2011 - 2015): Initial implementation of DXRAM: Metadata management and overlay, local memory management, Ethernet network subsystem
- **Kevin Beineke** (2013 - 2018): Logging and Recovery, DXNet and Ethernet transport
- **Stefan Nothaas** (2015 - 2019): Initial implementation of DXGraph, DXMem, DXNet, Infiniband transport, IBDXNet
- **Philipp Rehs** (2016 - *): DXGraph and applications
- **Filip Krakowski** (2018 - *): Migration, Dynamic up- and downscaling
- **Fabian Ruhland** (2018 - *): Graph based applications, IBDXNet (RDMA)
- **Michael Schoettner**: Chairholder and supervisor of DXRAM project



Introduction


- Data Growth
- HPC, Big Data and Graph Application Domains
- Application Requirements
- Challenges & Approach

Need answers quickly and on bigger data

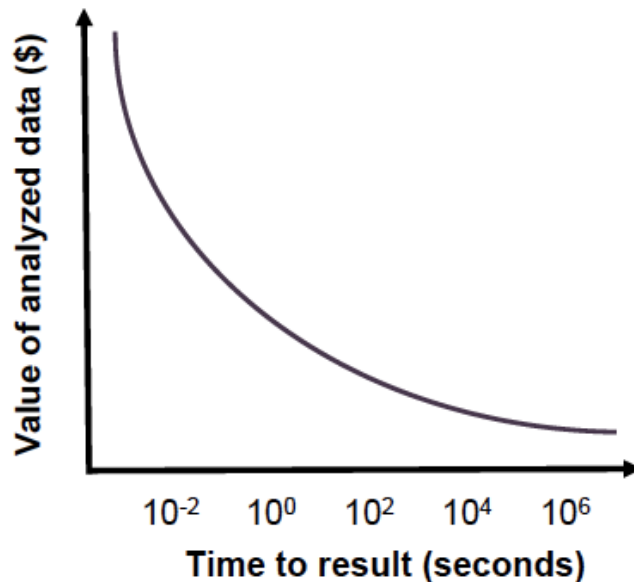


By 2020:

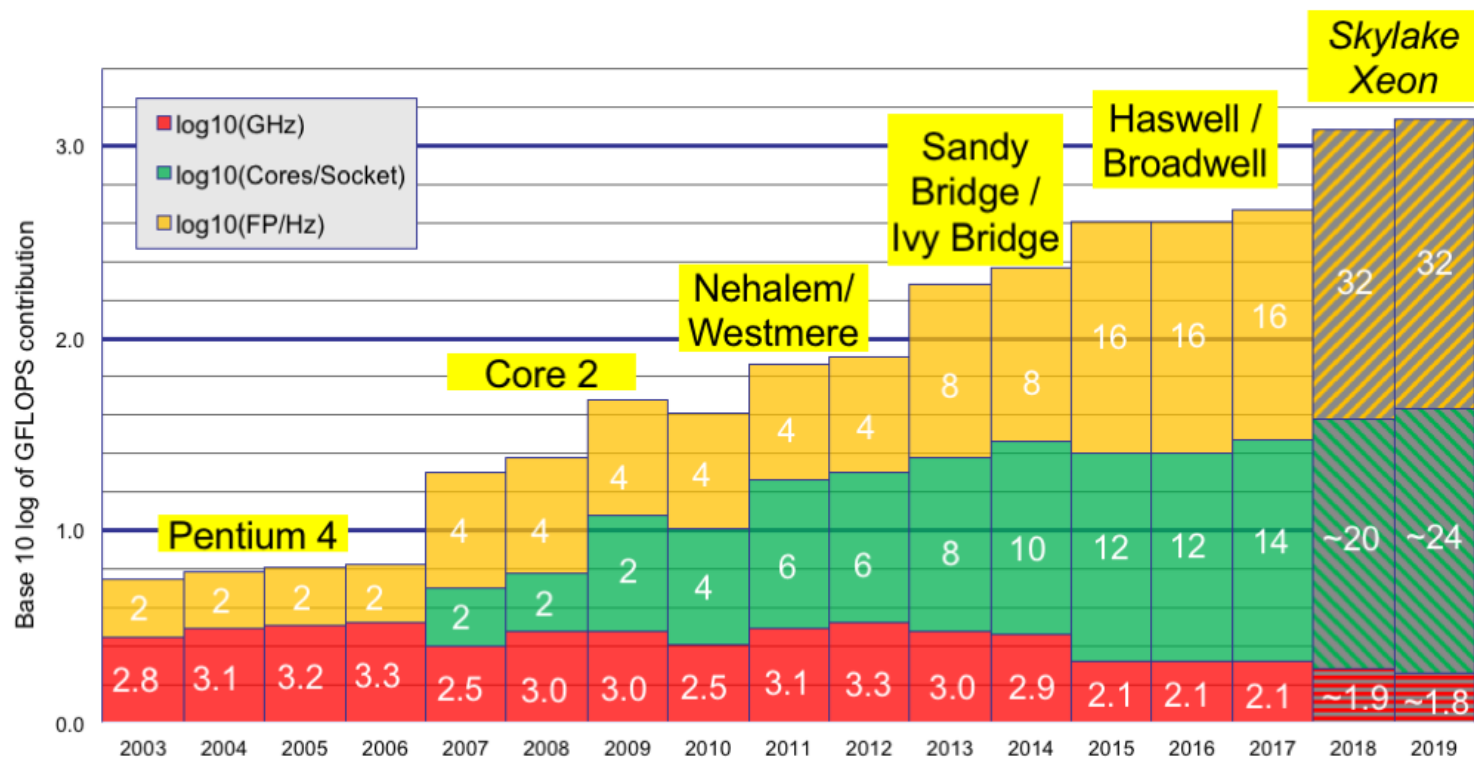
8B 
people

20B 
connected devices

100B 
infrastructure devices



Computing power is growing ~50% each year



Dr. McCalpin, University of Texas, Austin

High performance computing (HPC)

- Focusing on computing resources
- Aggregating many cores
- OpenMP: multithreading on each node
- Cuda: parallel computations on GPU
- MPI: exchanging data explicitly between nodes



Big data systems

- Designed to process huge amounts of data
- Aggregating many storage nodes

- Hadoop MapReduce:



- Batch processing of big files (tera bytes)
- Coordinate many parallel tasks executed on each node
- Only for embarrassingly parallel applications

- Apache Spark:



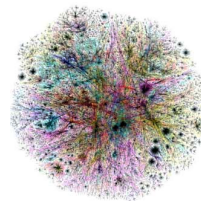
- Keep data in memory avoiding to write and read intermediate files to/from disk
- Resilient Distributed Data Sets: immutable; can be recomputed in case of failures



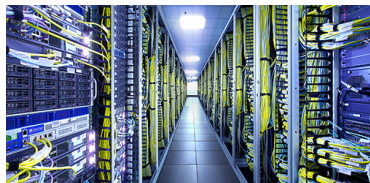
Graph Applications

- Interactive graph applications, e.g. social networks
- Online graph analytics, e.g. bioinformatics, WWW
- Online state management, e.g. for Stateless NFV

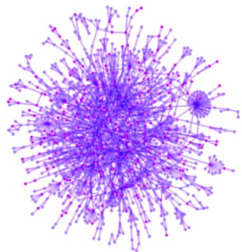
WWW



Network Function Virtualization (NFV)



Bioinformatics

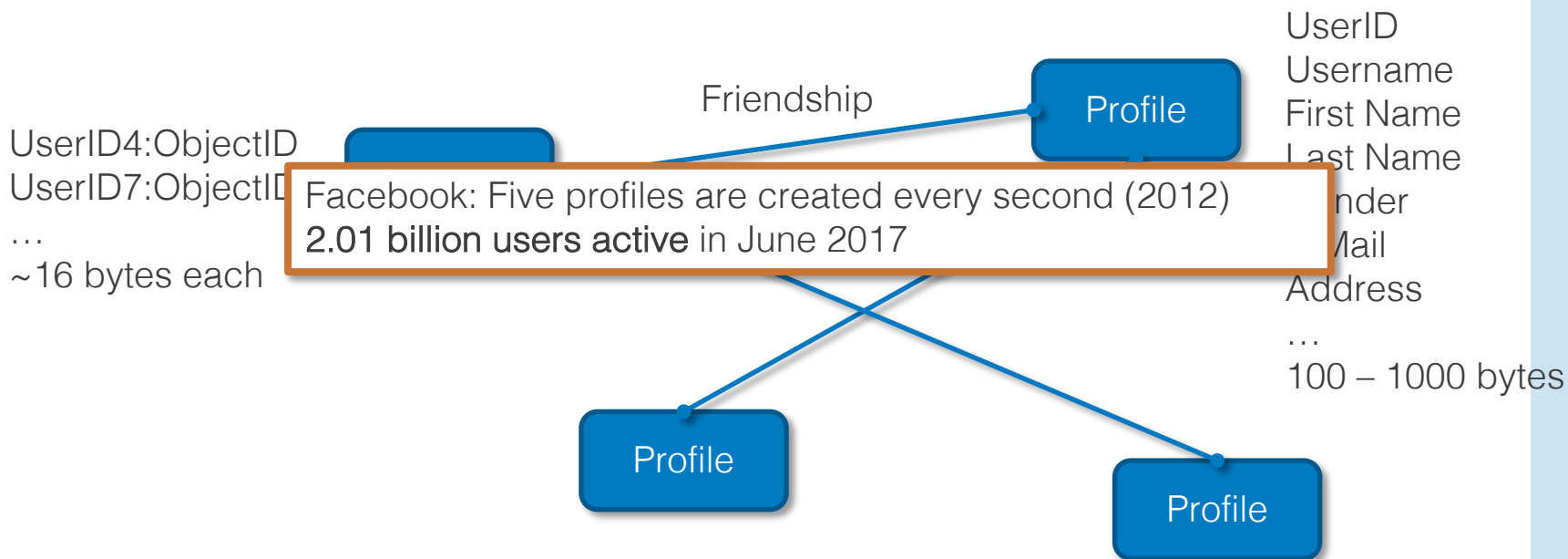


Social Networks



Interactive Applications: Social Networks

- Interactive requests from billions of users from the Internet to backend storage
- Irregular access patterns



Interactive Queries: Social Networks

RsrcID3:ObjectID
RsrcID1:ObjectID

...
~16 bytes each

Profile

Resource

Facebook: ~50,000 Likes, ~4,900 status updates and
~55,000 shares generated every second (2012)

70% of all data objects are smaller than 64 byte (2011)

Resource

Types:

Objects (< 100 bytes)

- Status updates (< 100 bytes)
- Likes (~16 bytes)
- Shares (~16 bytes)
- Friend requests (~16 bytes)
- Thumbnails (~100 bytes)
- ...

Online Graph Analytics: Bioinformatics

- Enumerating common molecular substructures
 - 156.000 smaller molecular graphs
 - Identify maximal cliques in product graph
 - Many graph comparisons (up to 156.000^2)
- Many small data objects (32 – 64 bytes)
- Requiring computations on storage nodes

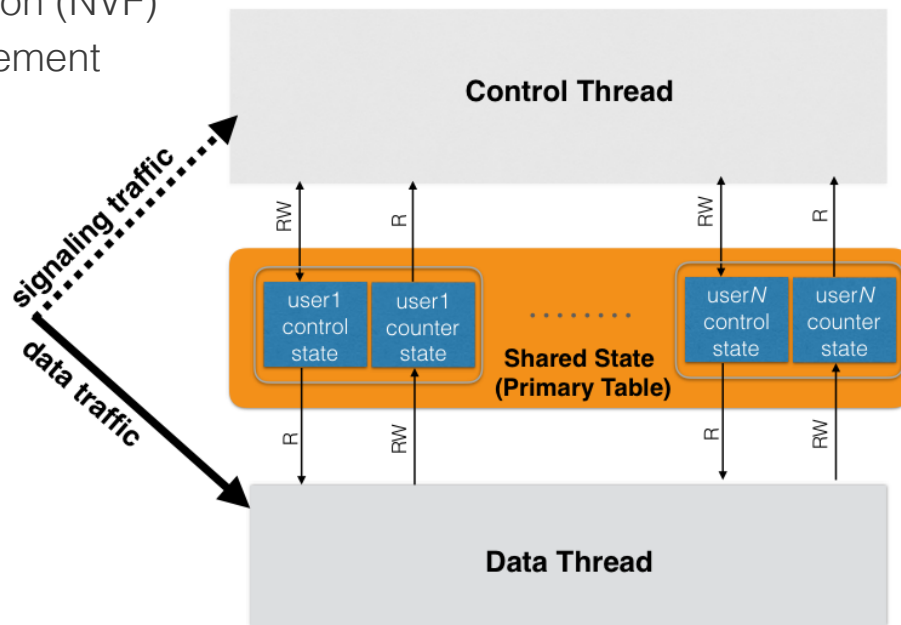


<https://www.cs.hhu.de/lehrstuehle-und-arbeitsgruppen/algorithische-bioinformatik/publikationen.html>



Online State Management: Stateless EPC

- Use case for Network Function Virtualization (NVF)
- Evolved Packet Code (EPC) state management for mobile users in shared store
- Low-latency data access to many small data objects is important
- Also requires a subscription and notification mechanism
 - E.g. if control thread changes user control state the store needs to send a notification to a remote data thread



Z. Qasi et.al.: A High Performance Packet Core for Next Generation Cellular Networks (SigComm 2017)



Summary of Graph Application Requirements

- **Low-latency data access** is very important
- Efficient management of **billions to trillions of small data objects**
 - Reads dominate (writes and deletes are seldom)
 - Data objects are small (32 – 64 byte per object)
 - Often irregular access patterns
 - High concurrency
- Some applications need to **run computations on storage nodes (locality awareness)**
 - Vertex-centric programming model (coordinated by super steps)
 - Graph-centric model requiring fine-grained synchronization



But I/O is the new bottleneck

- Low-latency is a must for interactive applications

Memory technology	Access Latency		Max. Throughput	
	Read	Write	Read	Write
Hard disk	9ms	9ms	112 MB/s	45 MB/s
NAND flash	47us	15us	3.0 GB/s	2.6 GB/s
DRAM	51ns	51ns	13 GB/s	13 GB/s
SRAM	1-2ns	1-2ns	>100 GB/s	>100 GB/s

<https://www.extremetech.com/extreme/211087-intel-micron-reveal-xpoint-a-new-memory-architecture-that-claims-to-outclass-both-ddr4-and-nand>



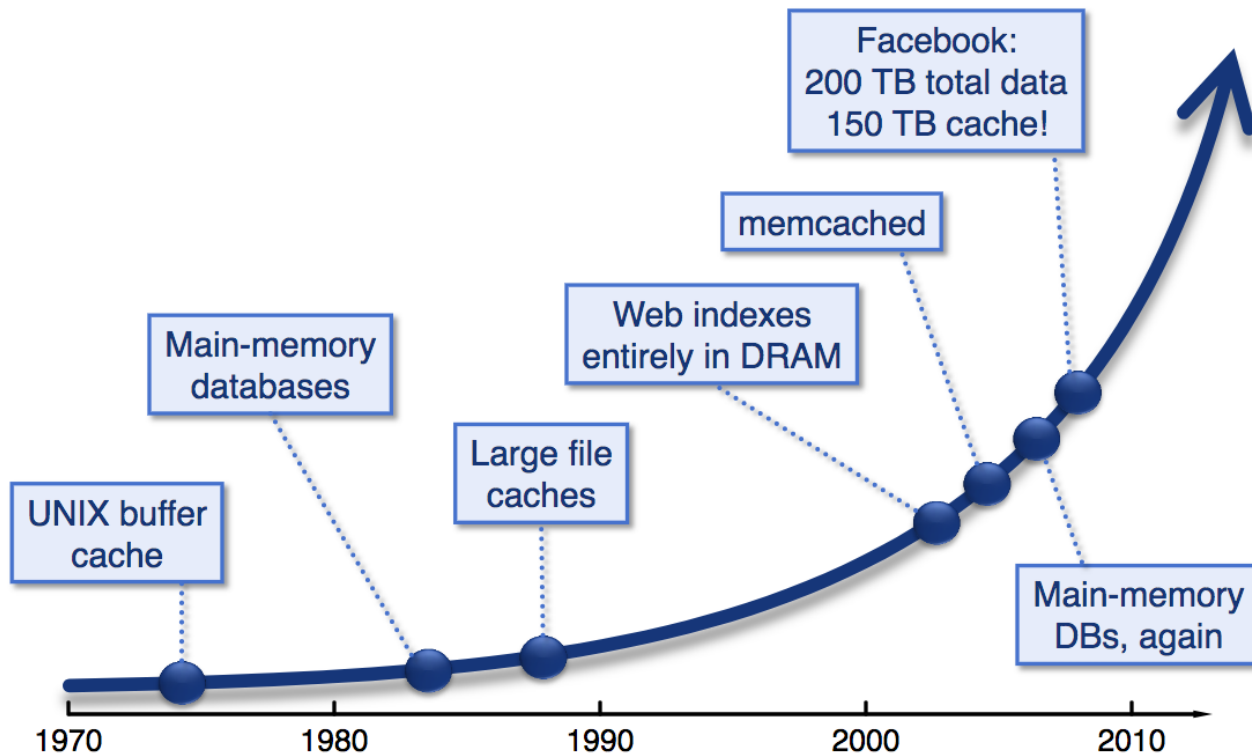


Distributed approach is a good choice

- Limitations of a single machine
 - RAM is still less than traditional (slow) storage
 - Number of cores limited
 - Increasing costs with such “fat” nodes
- A (highly) distributed solution is better than one (or a few) “fat” nodes
- FLOPS are increasing each year because of the growing number of cores on each socket
- But memory latency of servers is getting slightly worse each year
 - The reason is the bus bottleneck shared by all cores
 - And cache snooping between all cores



DXRAM in Storage Systems



The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM, Osterhout et.al.

Caching is not sufficient

- High cache hit rates needed because of the large access time gap between DRAM and disk
 - Even a 1% miss ratio for a DRAM cache costs a factor of 10x in performance
- Graph applications have **irregular access patterns** → **caches need to be very large**
 - Facebook keeps around 75% of all data always in caches
 - Used up to 1.000 Memcached servers for caching
- Caches need to be **manually synchronized** with backend storage to avoid data loss in case of server crashes or power outages

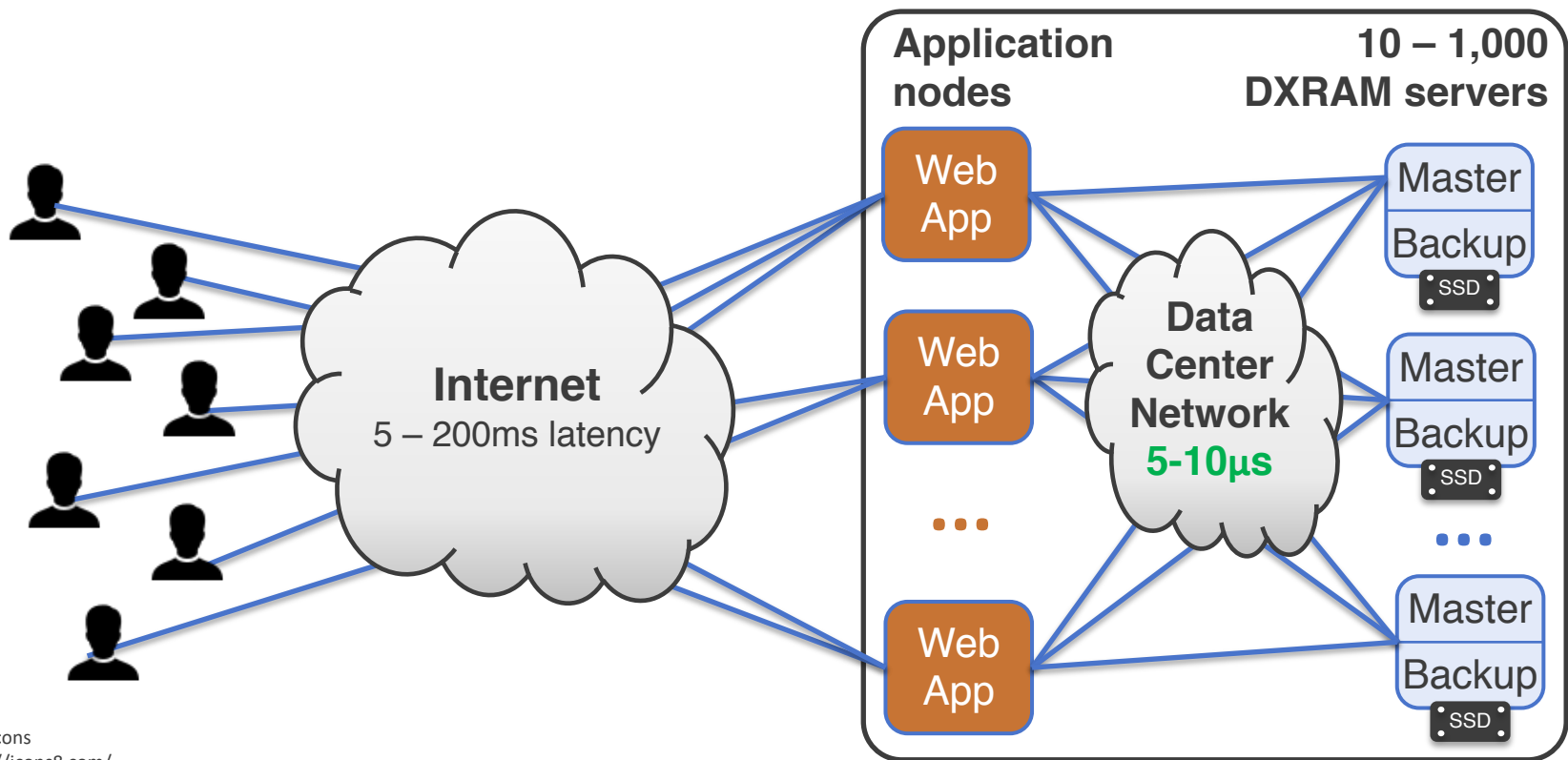


DXRAM - Distributed in-memory key-value store

- Challenges & Objectives



DXRAM: As a fast backend key-value store



User icons
<https://icons8.com/>

All data always in memory

- Aggregate tens to thousands of servers in a cloud data center as needed
- High-speed networks (10 – 100 Gbit/s) allow fast access to all data
- Data is not replicated in remote memory
 - Data volume would be multiplied → DRAM is expensive
 - Move computations not data → functions are much smaller than data volumes
 - Avoid consistency issues
 - Weak consistency scales but is complex
 - Strong consistency is great but does not scale
 - Does not help for fault tolerance in case of power outages



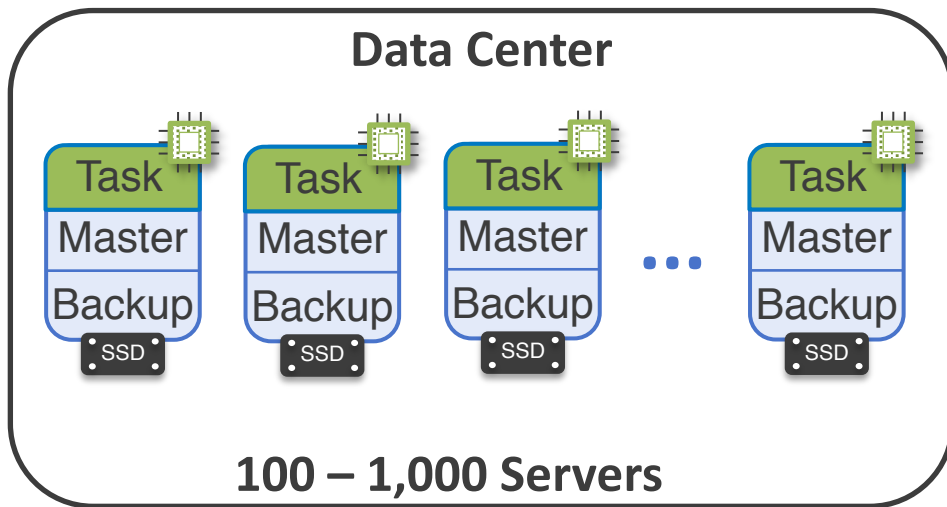
Fault tolerance

- **Crash-recovery model**
 - Single or several servers crash (likely)
 - Data center power outage (very seldom)
 - Network may be partitioned (very seldom)
- **Data is replicated on remote SSDs**
 - Remote replication allows immediately recovery
 - Persistence in case of data center power outage
- **Fast parallel server recovery (1 - 2 sec.)**
 - Minimizing impact for applications
 - While avoiding overhead of in-memory replication



Interactive compute platform

- Complement fast storage (DXRAM) with computations on storage nodes (DXGraph)
 - Distributed and parallel computations
 - Move tasks not data
- **Example:** enumerating common molecular substructures
 - Identify maximal cliques
 - Many parallel comparisons of many small graphs



Computation framework

- **Data models** based on key-value tuples
 - **Graph data model:** vertices and edges are stored as data objects
 - **Tables/Sets:** stored as tablets/subsets
- **Distributed task management:** move tasks not data
- **Application-controlled synchronization**
 - Barriers
 - Event notifications
- **Hot-spot migration:** for load balancing
- **Dynamic up- and down-scaling** of compute resources (nodes)





Summary - Challenges & Objectives

- Global and local lookup for **billions of small objects (32 – 64 bytes)**
 - **Metadata explosion:** Storing the object location of every single object is very expensive
 - A single lookup server is hazardous -> **distribution**
- **Distributed management** for billions of small objects
 - **Low remote latency (~10 us):** High speed network
 - **Low local latency (~1 us):** Low latency storage -> **RAM**
 - **A low overhead:** 5 – 10% metadata overhead in RAM
 - **High concurrency (100's of threads):** Minimize locking/lock free -> Millions of ops
- **Dynamic up- and down scaling:** Data is evolving and expanding, thus the cluster must scale as well (1000's of servers)
- **Reliability: Persistency** (→ power outage) and **fast recovery** -> single failed server < 2 sec
- **Dynamic data migration** and load balancing -> some profiles/resources are very popular (zipfian distribution).



DXRAM - Core Architecture

- Architecture
- Node Types & Overlay
- Metadata Management
- Memory Management: CID Translation & Small Object Heap

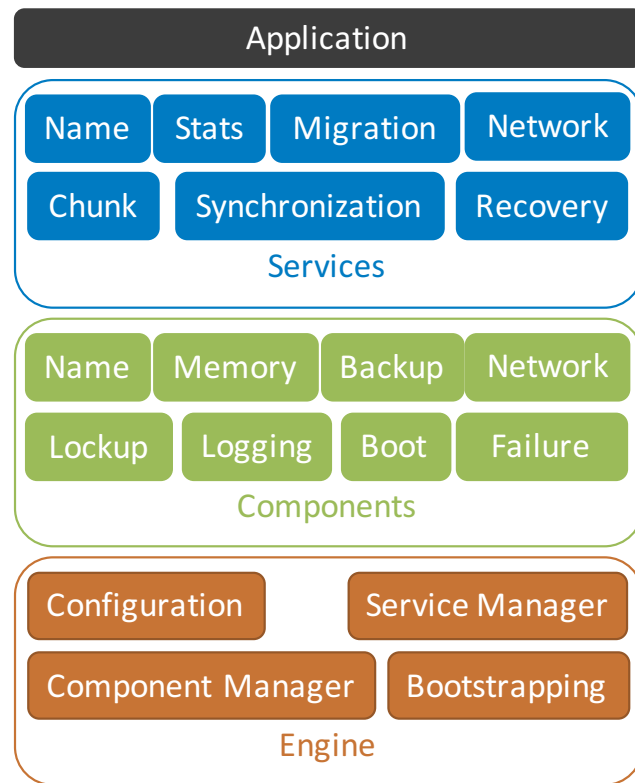
DXRAM – Key Facts

- **Distributed system** for clusters in data centers.
- All data always in RAM.
- **Key-value** data tuples: “chunks”
 - Key: 64 bit globally unique sequential chunk ID (CID)
- Optimized for handling **billions of small chunks**.
- **High throughput and low latency** networks: 10 Gbit/s Ethernet and 56 Gbit/s Infiniband
- Persistency through **logging** to raw device (SSD aware).
- Parallel distributed **fast recovery**.
- **Dynamic up- and downscaling** of cluster.
- **Data migration** to handle hot-spots.
- Written in **Java**.
- **Open source** at Github.



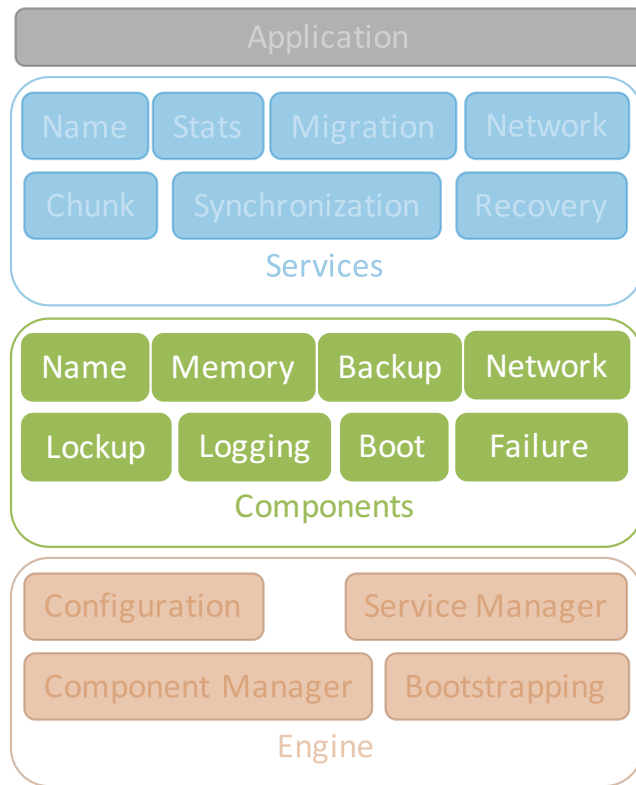
Architecture

- Modular software stack
- **Engine**: Management tasks for modules
- Core functionality implemented as **components** and/or **services**
- **Services** form **API** for applications
- Applications implemented as **DXApp** (loadable jar package)



Components

- Implement **node local** functionality/features
 - Access to hardware (native memory, disk, network)
 - State store and management
 - Caches
- Components can **access other components** for data exchange
 - Access to own node ID from boot component
- Modularization: Split into multiple components
- Can be **enabled/disabled** if not needed → save resources
- Can be limited to node type: superpeer/peer



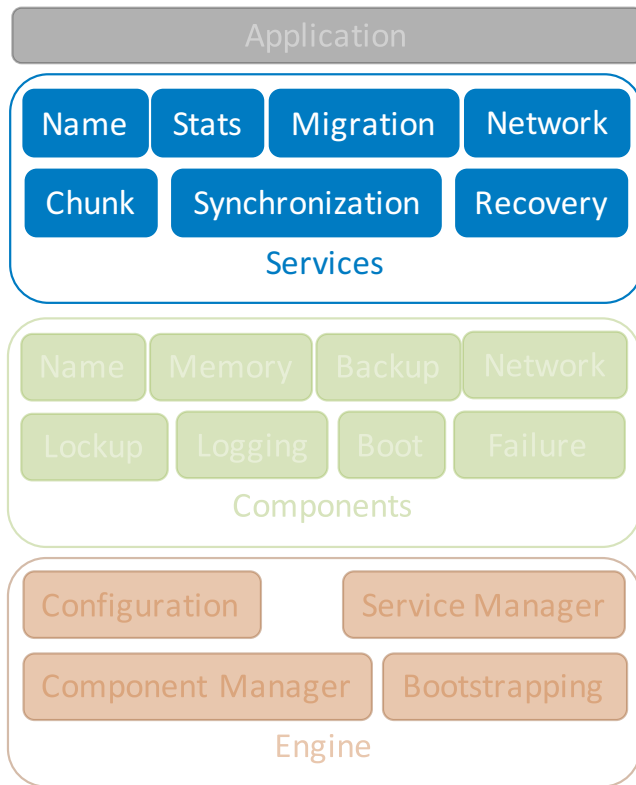
Components

- **Application:** DXApp package management for applications running on DXRAM nodes
- **Backup:** Management of backup ranges and backup tree
- **Boot:** Node bootstrapping, Node ID assignment, Node mappings
- **Chunk, Chunk Backup and Migration:** Access to local DXMem backend storage
 - CRUD operations for storing application chunks
 - Special components for fast backup and migration
- **Event:** Local event signaling and handling system
- **Failure:** Node failure handling
- **Job:** System running worker threads executing queued jobs
- **Log:** Access to DXLog for logging (persistent data backup) with access to disk
- **Lookup:** Overlay management and lookup cache tree
- **Name:** Naming index structure
- **Network:** Access to DXNet for sending and receiving messages



Services

- Implement (high level) **API** for applications (DXApp)
- Handle **communication** with same service on **remote node**
- Isolation: One service cannot access other services → Avoid dependencies on API
- Can be **enabled/disabled** if not needed → save resources
- Can be limited to node type: superpeer/peer
- A component does not require a matching service and vice versa



Services (API)

- **Application:** Running DXApp jar packages, one (main) thread per application
- **Boot:** Expose own node ID, node ID mappings, superpeer/peer list, ...
- **Chunk:** Various services offering different types of operations
 - CRUD operations for chunks
 - “Anon” operations for chunks without knowing the type (byte[]) data used)
 - Debug for memory debugging
 - Chunk pinning and direct memory access
- **Job:** Enqueue new jobs either locally or to remote nodes running the job service
- **Log:** Access local and remote log (writing backup data to disc) information
- **Logger:** Access the local or a remote (text) logger (e.g. set log level)
- **Lookup:** Access to the (remote) superpeer overlay, lookup tree, metadata, ...
- **Migration:** Migrate chunk(s) from one peer to another one



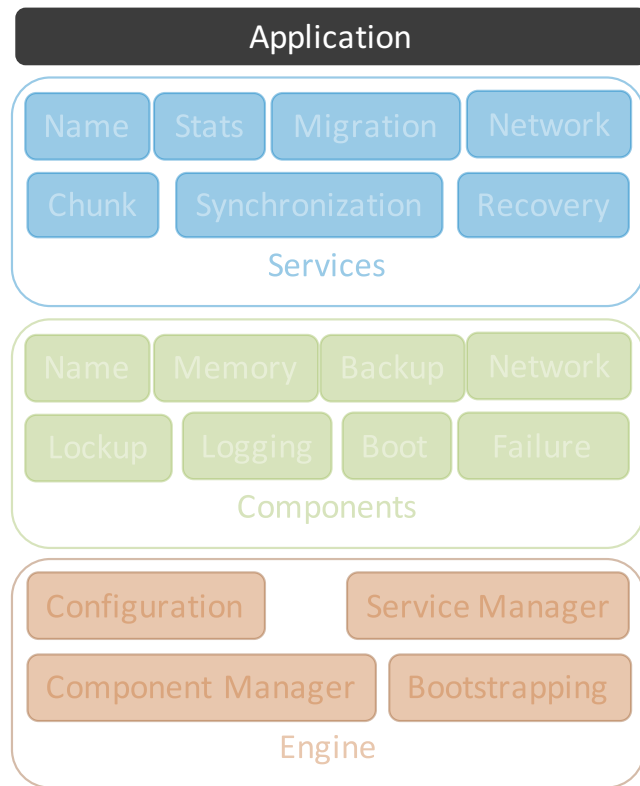
Services (API)

- **Master Slave Compute:** Task based computations running on peer nodes
- **Network:** Send and receive messages
- **Recovery:** Handling of remote recovery messages
- **Statistics:** Access to statistics collected in various components and services
- **Synchronization:** Barrier synchronization for computations on peers
- **Temporary Storage:** Small and chunk-store independent scratch pad (no persistency)



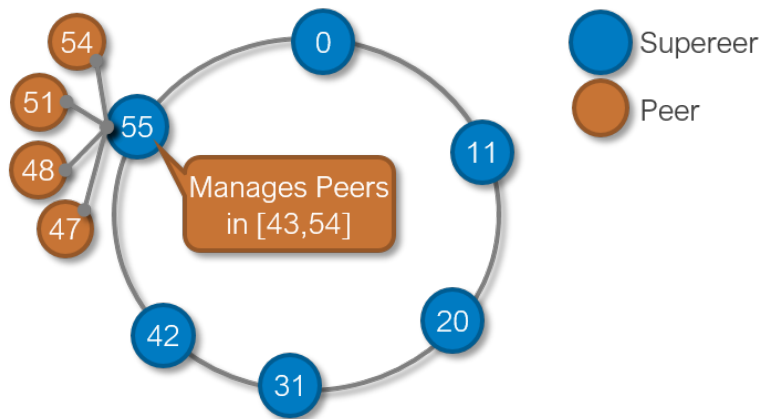
Application

- Implement application interface → **main-Method**
- Access to **API** (Services)
- Compiled as separate jar-Package
- Loaded by DXRAM (Application Service) on boot
- Run multiple and different applications on peers
- DXRAM applications
 - Benchmarks
 - REST-API (server)
 - Terminal (server): For CLI to access services
 - DXGraph: Graph framework



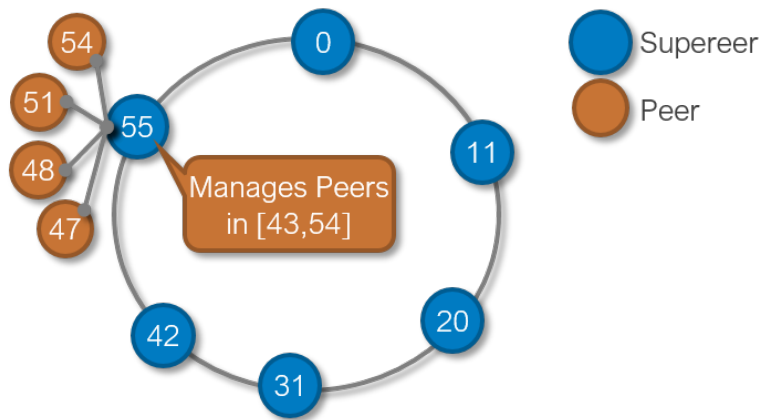
Node Types - Superpeers

- Store **global meta-data**:
 - the locations of chunks
 - nameservice entries
 - monitoring data
 - node states
 - backup distribution
- Coordinate recovery.
- Form **Chord-like overlay** $\Rightarrow O(1)$ node lookup
- Meta-data replicated on successors.
- 5 to 10% of all servers are superpeers.



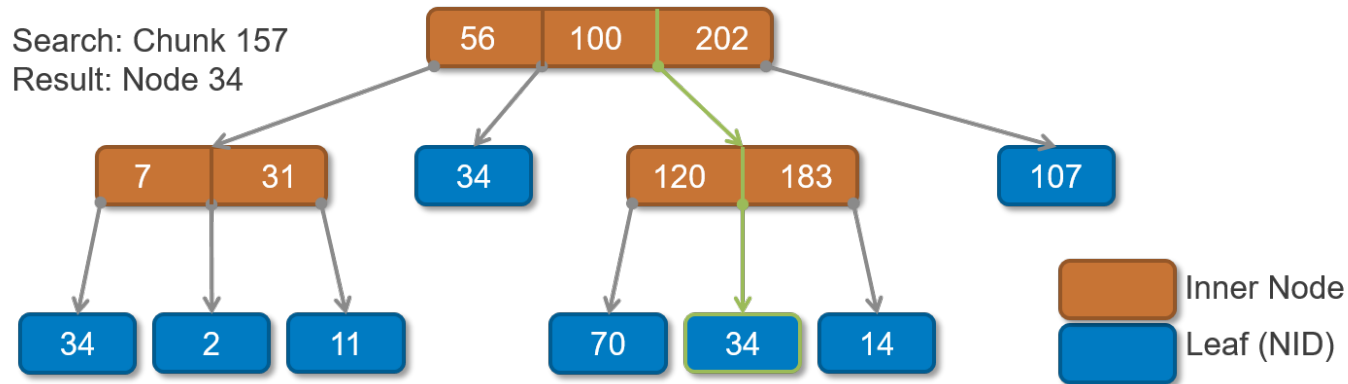
Node Types - Peers

- Every peer is assigned to one superpeer.
- Roles:
 - *Storage server*: store **data (chunks)**.
 - *Backup server*: store **backup data** of other peers (optional to all storage servers).
- May run computations and exchange data directly with other peers.
- Serve client requests when DXRAM is used as a back-end storage.



Global CID Lookup on Superpeers

- Lookup-Tree is stored on superpeers.
- Based on modified B-tree.
- Stores key-key-value tuples: beginning of range, end of range and location.
- Lookup in $O(\log n)$.
- CID aggregation \Rightarrow Low memory consumption for many chunks.



DXMem - Local Memory Management

- Chunks
- Memory Management
- Custom allocator
- Chunk Lookup



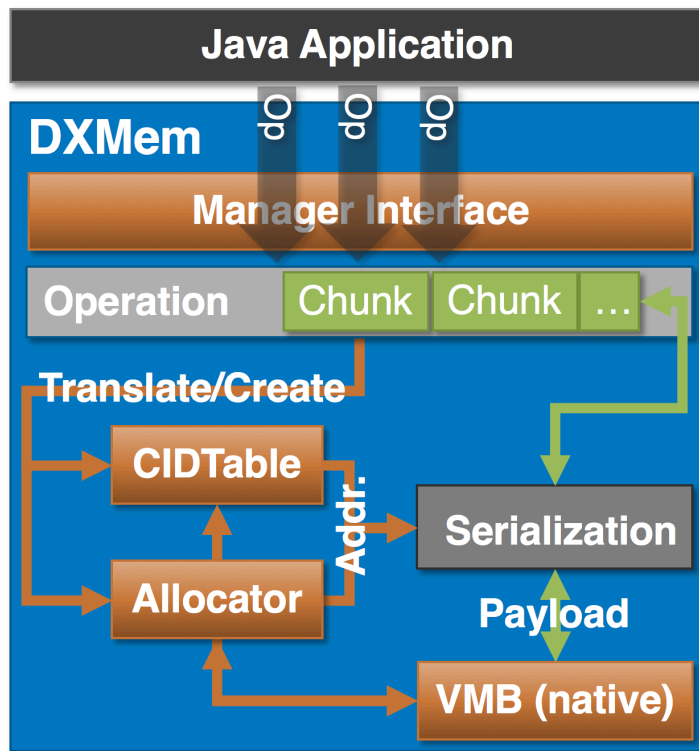
Objectives and Challenges

- Highly interactive applications => low-latency data access
 - Data kept in-memory: HDD/SSD slow
 - High concurrency is the rule
 - Irregular access patterns
 - Reads dominate (writes and deletes are seldom)
 - But: Data races on concurrent updates
- Vast data volumes (billions/trillions of objects)
 - Data objects are small (16 – 128 byte per object)
 - Main memory is limited (64/128 GB per server) compared to HDD/SSD
 - Efficient memory management
 - Every byte counts: one billion objects, one byte per object wasted => ~0.93 GB
 - Keep high data locality => lowers latency
 - Data distribution to multiple servers required (e.g. replication, immense volumes)



DXMem

- Local memory management in Java
- All data stored in RAM
- Key-value data model
- Operations: **Create**, **get**, **put**, **remove**
- Optimized for storing small objects (**16-128 bytes**), outside of the Java heap
- Fast **O(1) lookup table** to translate keys to memory address, memory efficient
- **Consistency**: Per chunk read-write lock for fine grained concurrency control
- Concurrent memory **defragmentation** (optional)
- Pinning for **direct memory access** and RDMA



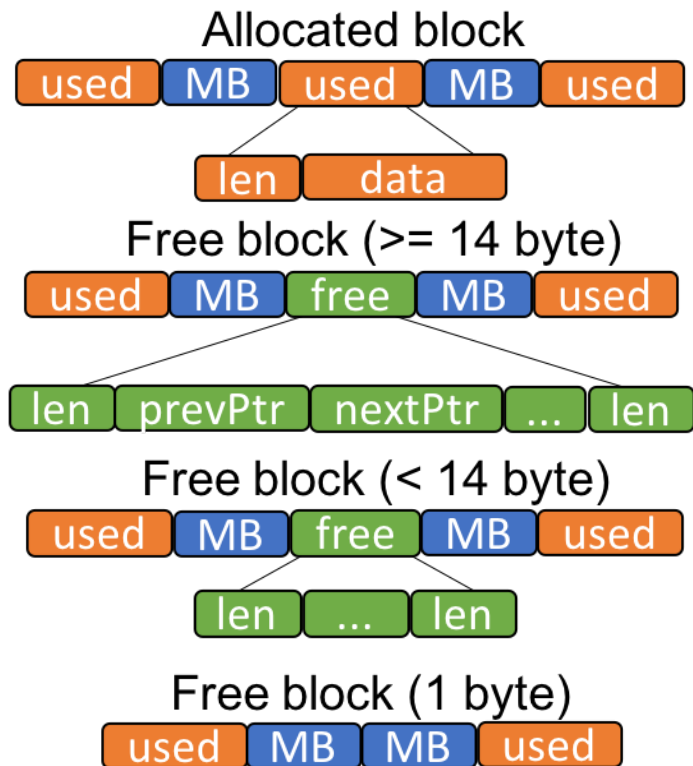
Chunks

- Chunk
 - 64-bit key ("chunkID"), binary data as value, key + value = "chunk"
 - Describes abstract concept: no type reflection on object
 - "Everything stored in-memory is a chunk"
- (Abstract) Chunk
 - Interface for implementing custom data types/structures in DXRAM
 - Serialization: Transparent for storing in-memory and network sending/receiving
 - No reflection or type information required/stored (if required: user can add own type information)
 - Generic "chunk" (wrapper for binary array) already implemented



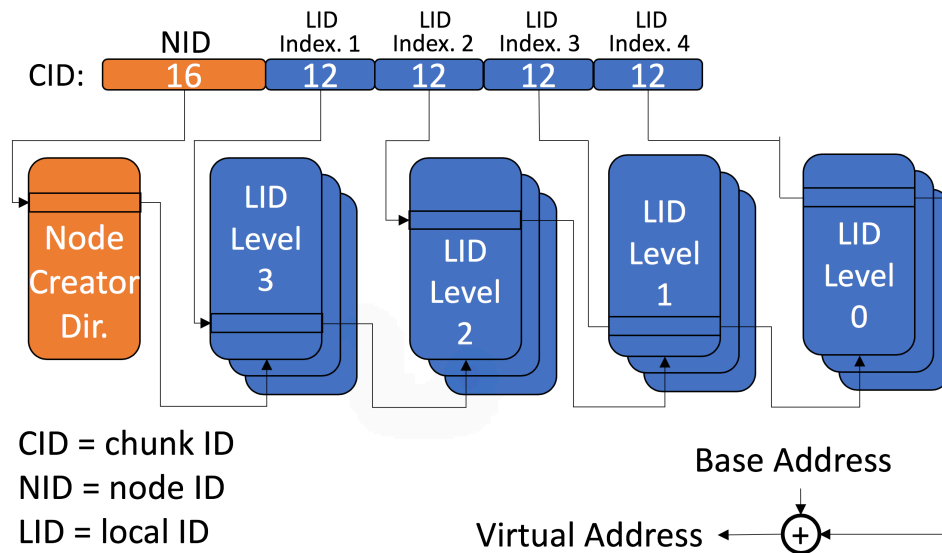
Allocator

- 43-bit pointers to address up to 8 TB of memory
- Allocation by best-fit strategy
- Custom serialization of Java objects to binary data
- "Marker byte" (MB): Block separator with metadata
- Compacted length field stored with block
- Free blocks managed as a doubly linked list



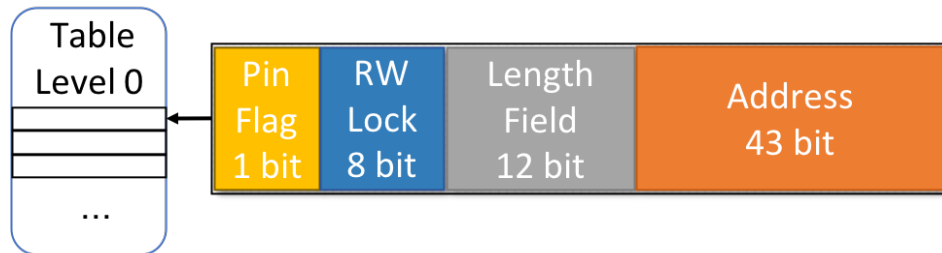
CIDTable

- No bare memory addresses: chunkID as key to access data
- CIDTable: translate chunkID to memory address similar to OS paging
- Fast (5 direct memory lookups)
- Tables created on demand
- Compact and memory efficient (re-use of chunkIDs)



Fine granular locking

- CIDTables
 - 64-bit alignment
 - Level 0 entry size: 8 bytes
- Memory efficiency: Split length field (for small chunks)
- Chunk pinning (e.g. for RDMA)
- Read-write-lock (CAS)
 - Fine granular: lock single chunk with data
 - Coarse granular: use empty chunk as lock

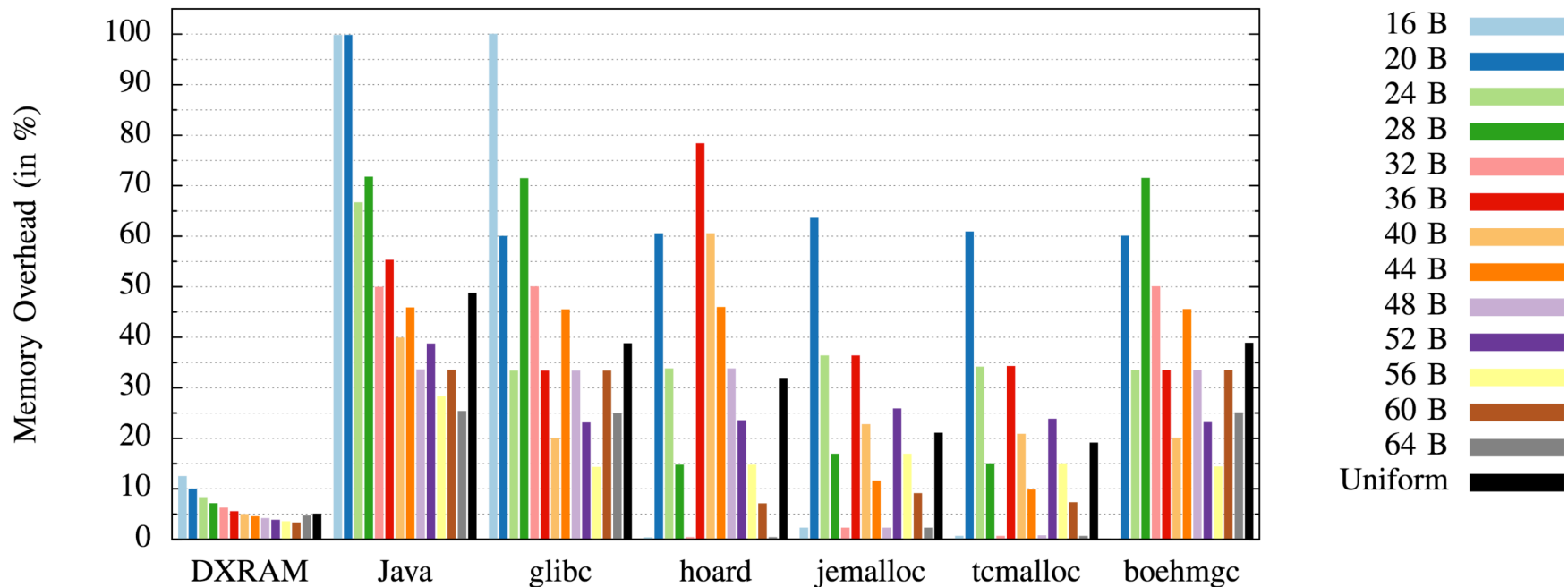


Evaluation (1)

- Compared DXMem allocator to other well known allocators regarding memory overhead
 - Java
 - glibc
 - hoard
 - jemalloc
 - tcmalloc
 - BoehmGC



Allocator Memory Overhead

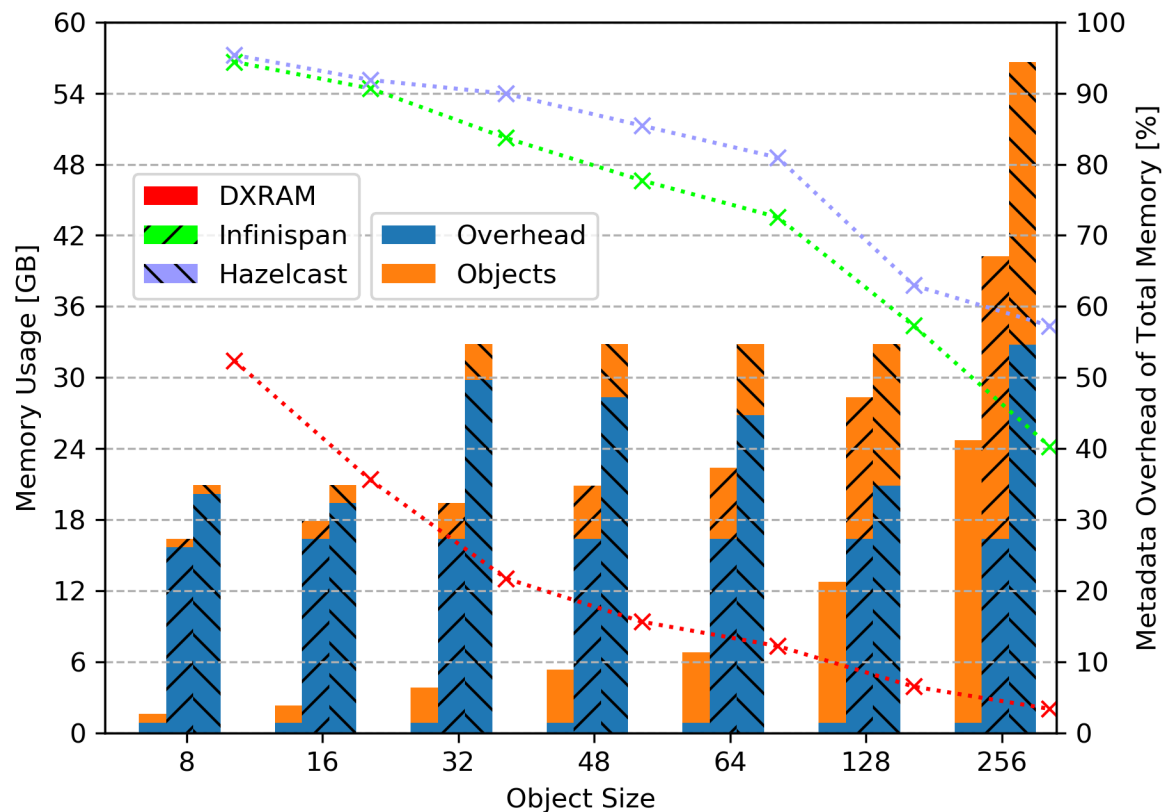


Evaluation (2)

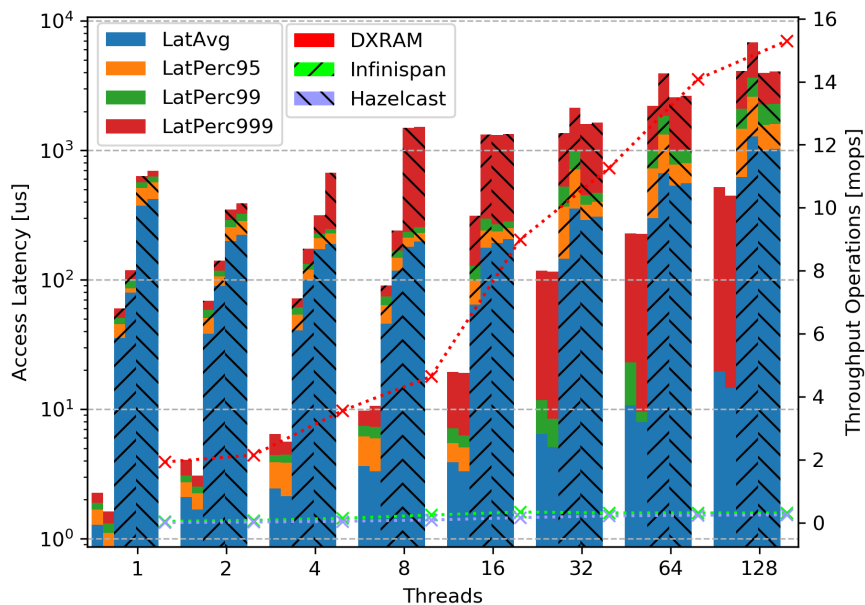
- Compared DXMem and DXRAM (using DXMem) to
 - Hazelcast with HD memory
 - Infinispan
- All systems store their data outside of the Java heap
- Benchmarks
 - Metadata overhead of local memory management
 - Concurrent local access with YCSB workloads
 - Distributed key-value store with YCSB workloads
- YCSB Workloads
 - YCSB-A: Objects with 10x 100 byte fields, 50% get 50% put
 - Facebook-B: Objects with 1x 32 byte field, 95% get 5% put
 - Facebook-D: Objects with 24x 32 byte fields, 95% get 5% put
 - Facebook-F: Objects with 1x 64 byte fields, 100% get



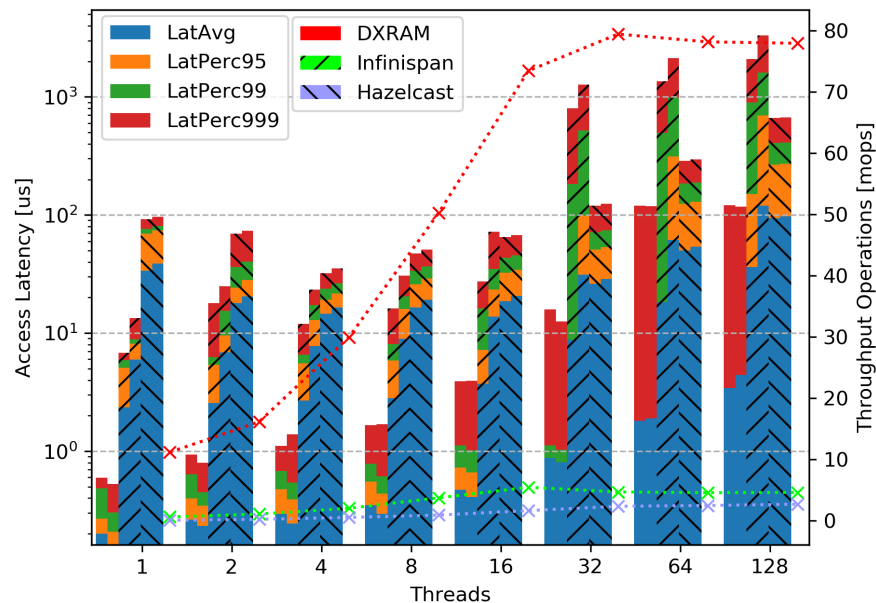
Local metadata overhead



Concurrent local access

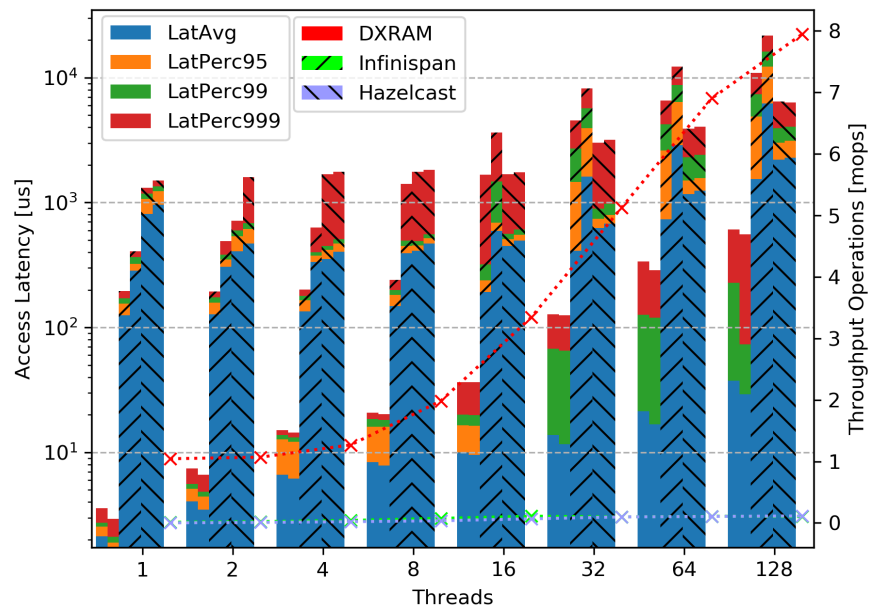


YCSB-A

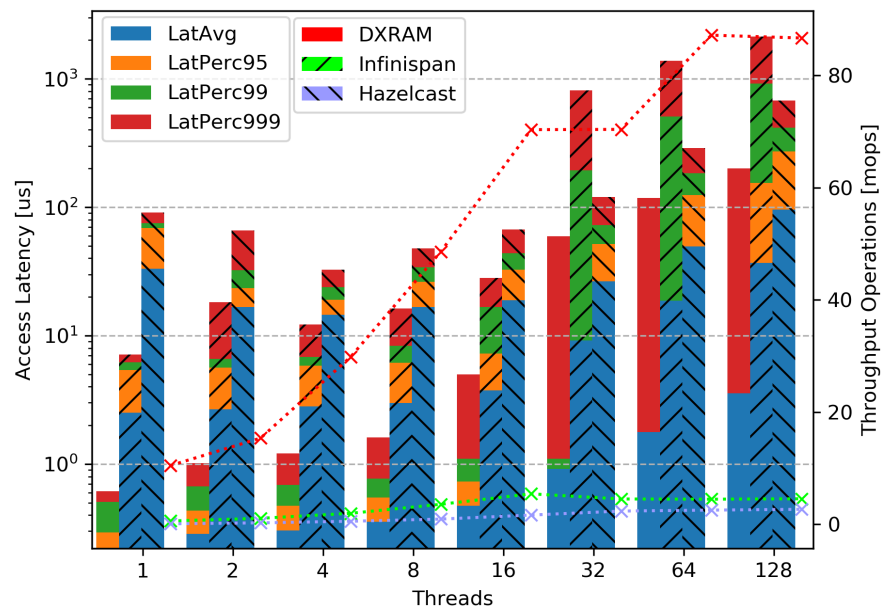


Facebook-B

Concurrent local access (2)

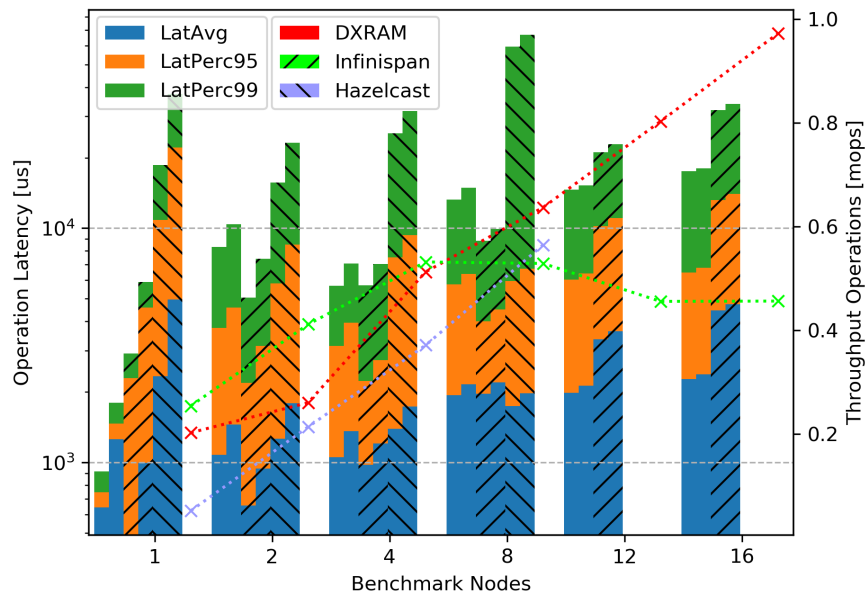


Facebook-D

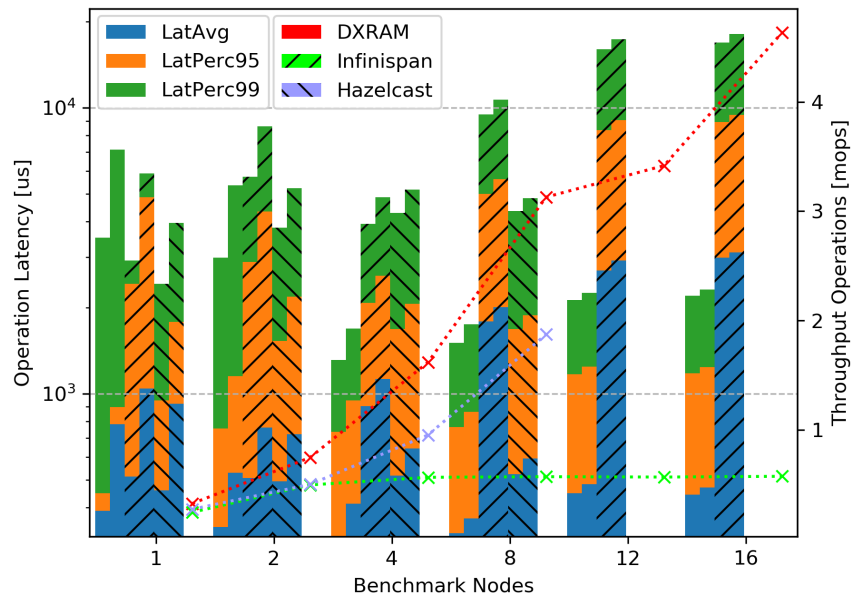


Facebook-F

Distributed key-value store

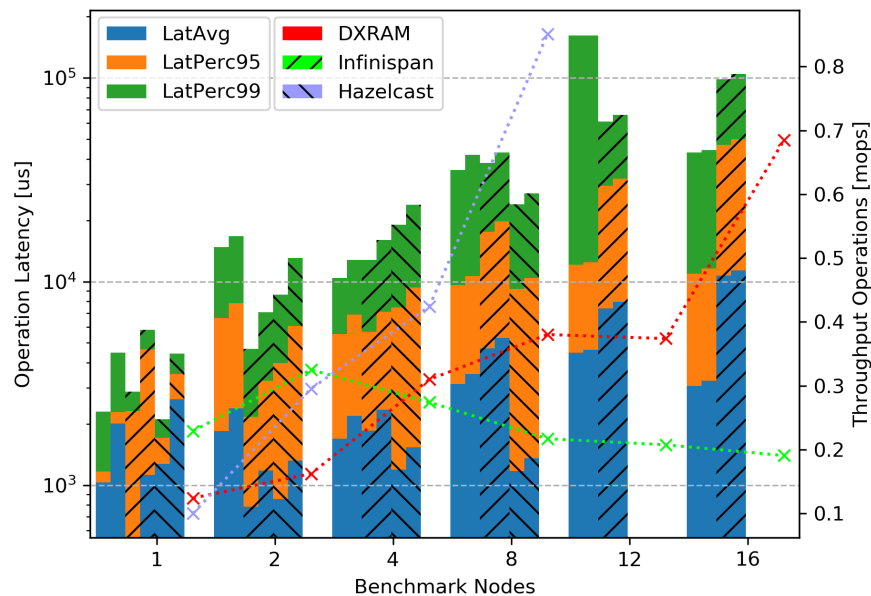


YCSB-A

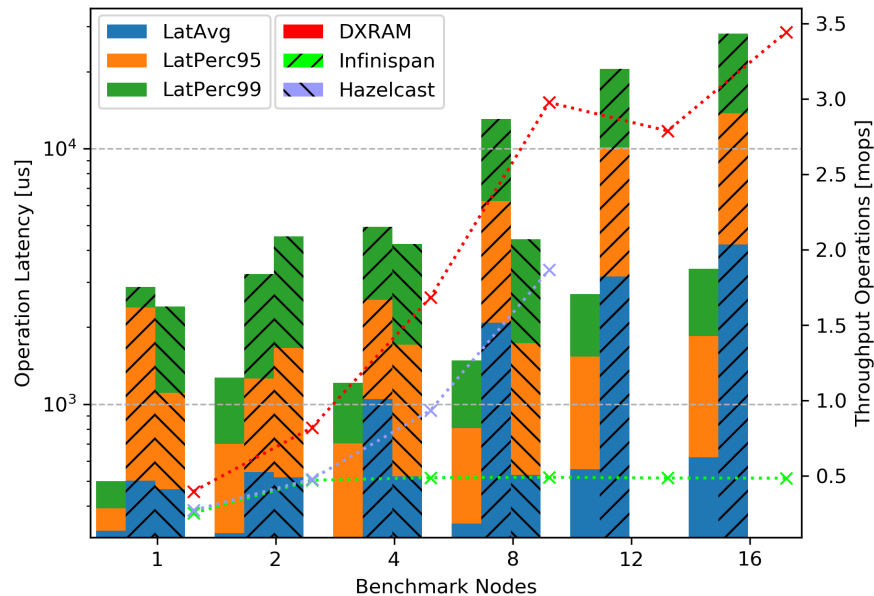


Facebook-B

Distributed key-value store (2)



Facebook-D



Facebook-F

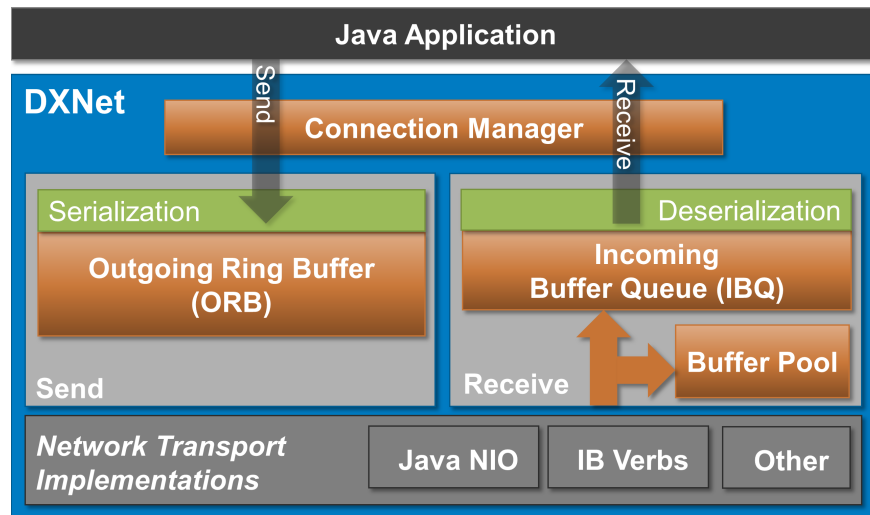
DXNet: Network subsystem

- Sending and receiving
- Transports
- Serialization: Importable/Exportable



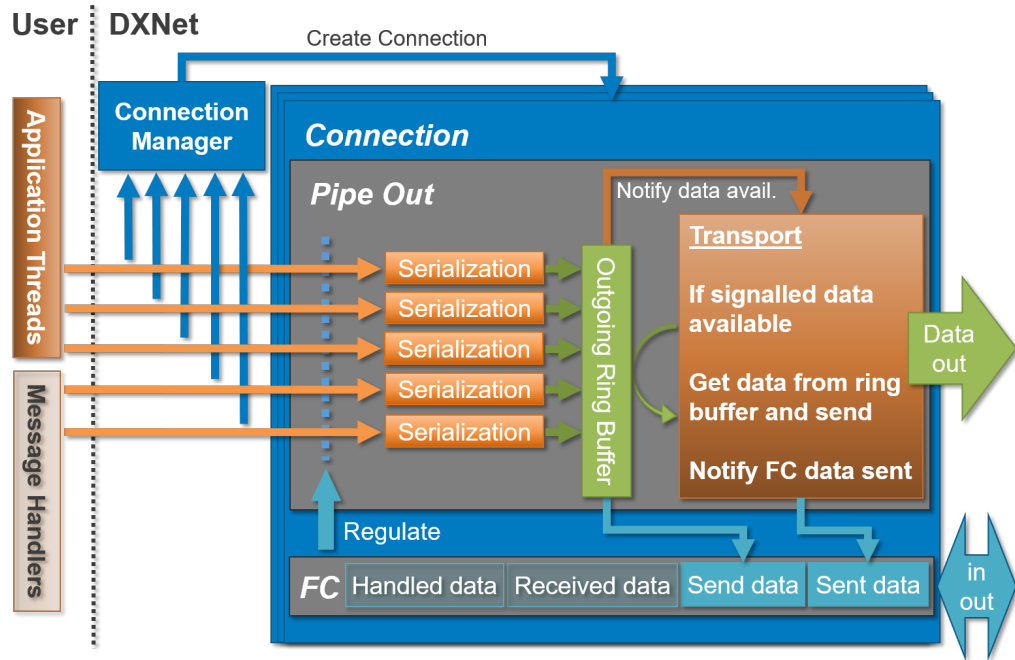
Architecture

- Network subsystem abstracts **transport layer**
- User can send **messages** and **requests**.
- Low latency and high throughput.
- Supporting: 10 Gbit/s Ethernet and 56 Gbit/s Infiniband.
- **Zero-Copy** (not counting de-/serialization of message *objects*).
- Flexible and highly concurrent **serialization**.
- **Lock-free** outgoing ring buffer supporting concurrent serialization into **native memory**.
- **Event-driven** message processing.
- Pipelining to increase throughput.
- Multi-level **flow control**.



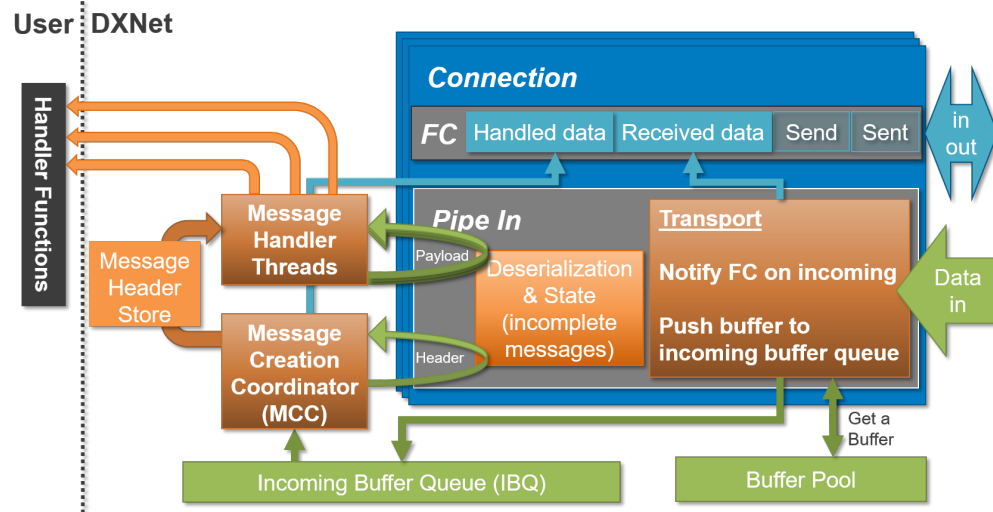
Sending of messages

- Optimized for many threads sending in parallel.
- Messages are aggregated in ORB without impairing latency.
- ORB lies in native memory
 - > sending without copying
- Message ordering is preserved.
- Lock-free implementation with low CPU load.



Receiving of messages

- Event-driven.
- Buffer Pool: contains buffers in three different sizes to be filled with incoming data.
- Multithreaded deserialization:
 - MCC imports message headers in order.
 - Handler threads create message objects and import payload in parallel.



Transports

- >10 Gbit/s Ethernet:
 - Based on Java NIO.
 - DirectByteBuffers are mapped onto the ORB -> sending without copying to kernel buffer.
 - Two channels per connection to avoid connection duplication.
 - Back channel is used for flow control messages.
- >56 Gbit/s InfiniBand:
 - Based on C++ library IBDXNet.
 - ORB is allocated in native memory; access with JNI



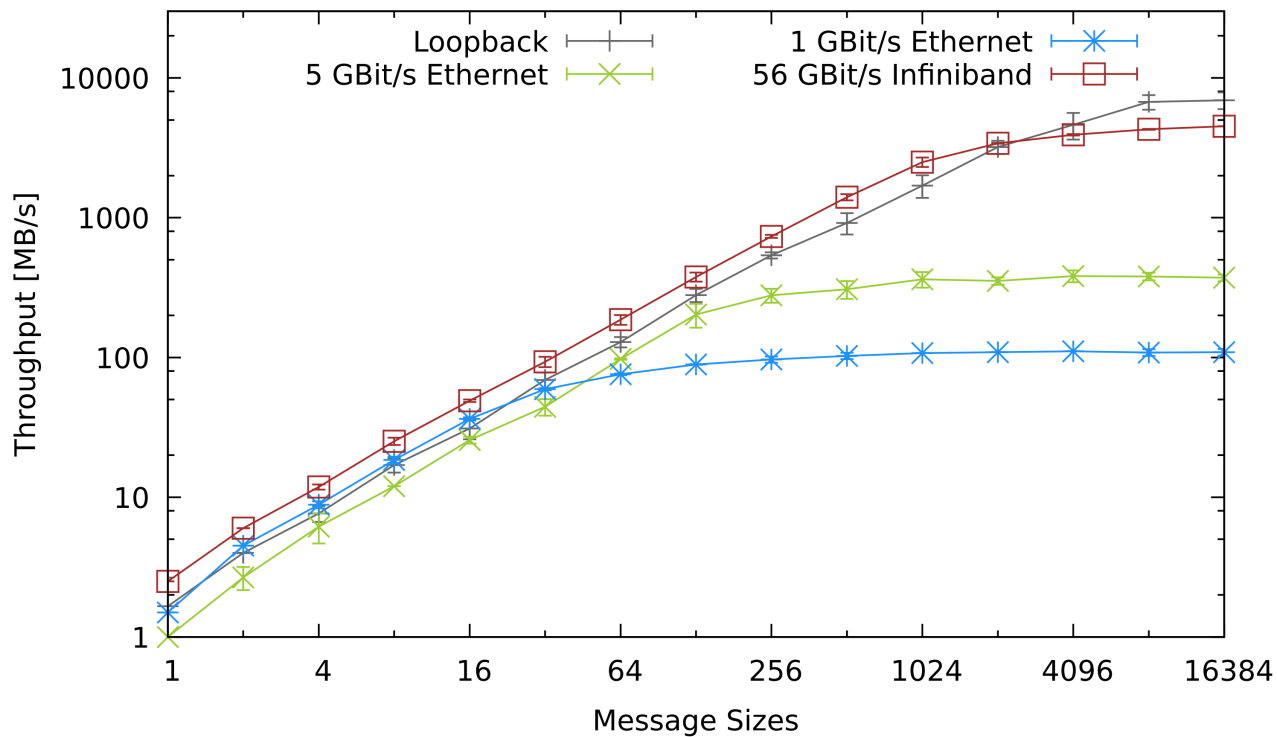


Importable/Exportable Message Data

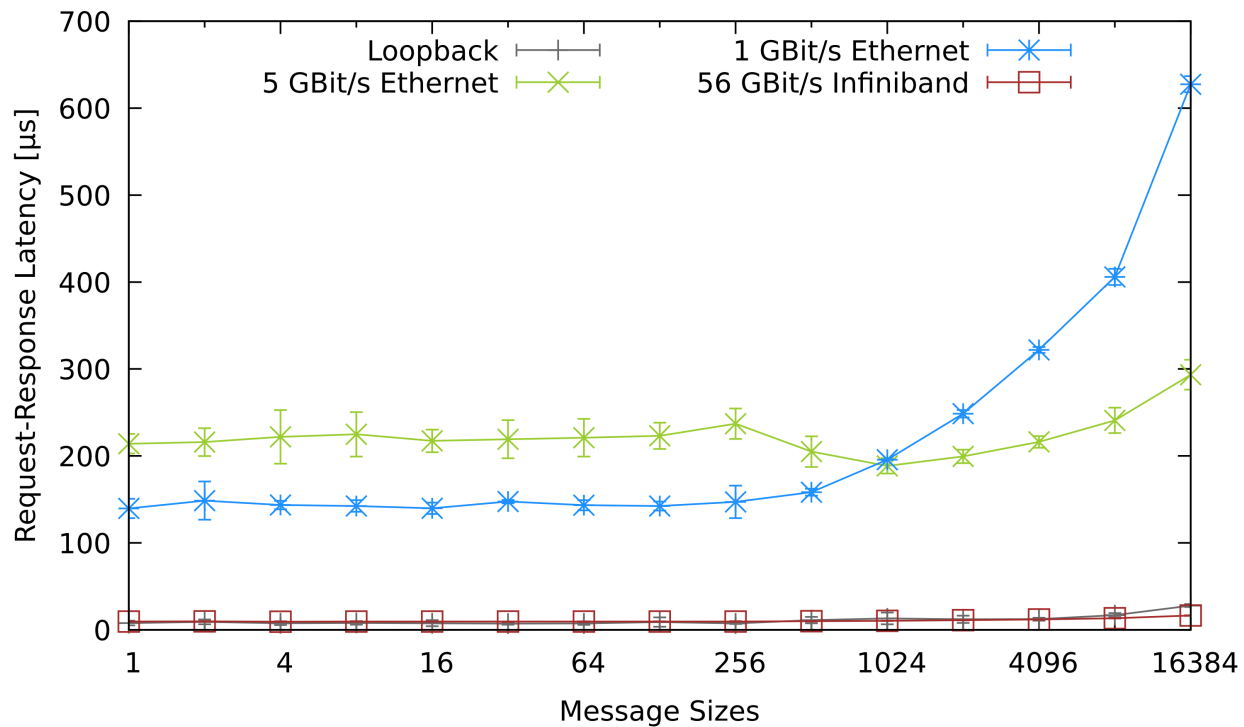
- Interfaces for messages:
 - Importable: How a message object is deserialized from an incoming byte stream.
 - Exportable: How a message object is serialized to an outgoing byte stream.
- Implemented by: Message, Request, Response
- Objects within a message need to implement the interfaces as well.
- User has control which data and how it is de-/serialized.
- But: De-/serialization process transparent to the user.
- Flexible and high performant serialization for network subsystem.



Evaluation



Evaluation





IBDXNet - InfiniBand Network subsystem and DXNet transport for Java

- InfiniBand in Java
- Sending and receiving
- Transports
- Serialization: Importable/Exportable

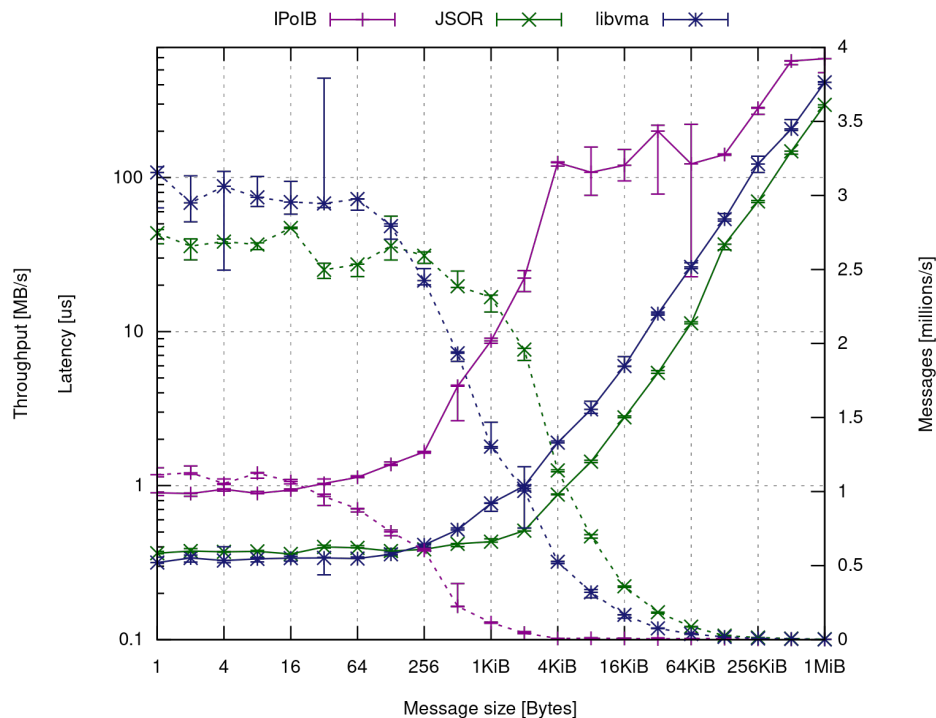
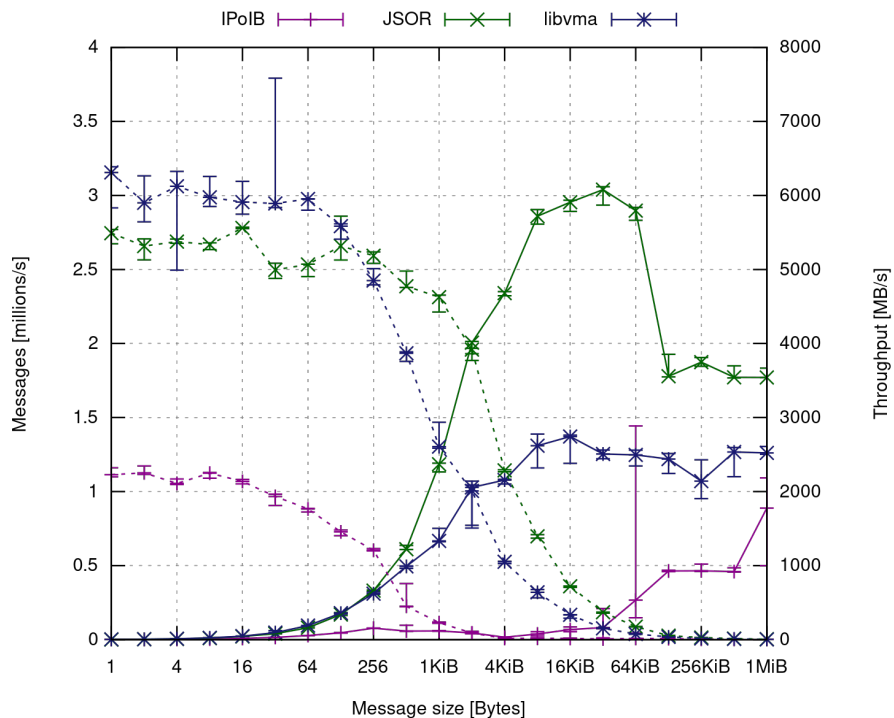


InfiniBand in Java

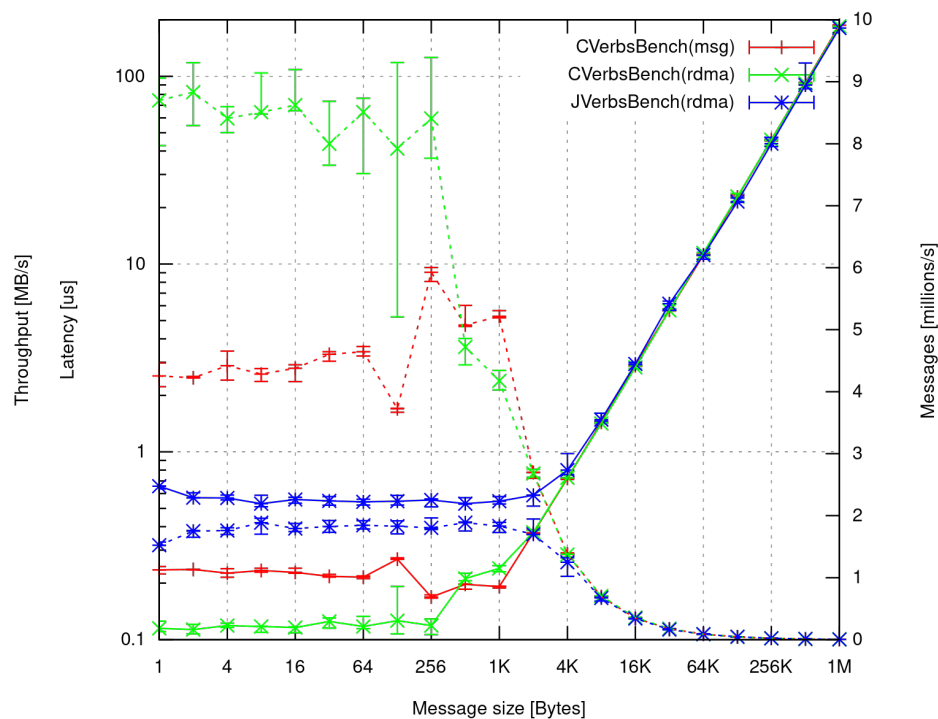
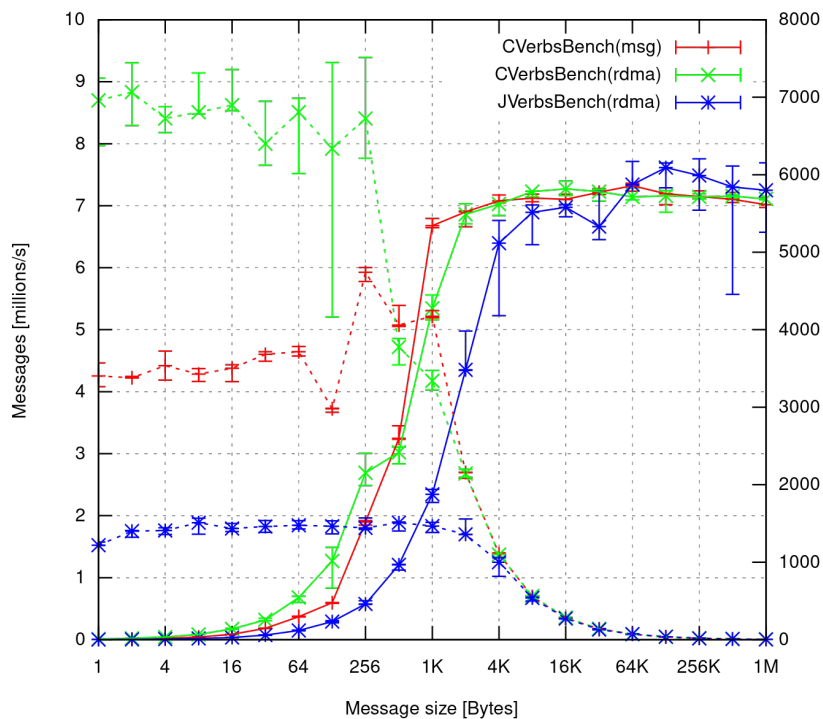
- Sockets programming vs. verbs
- Multiple options available for Java
 - IPIoverIB
 - libvma
 - Java Sockets over RDMA (JSOR)
 - jverbs (IBM JVM, only)
 - JNI + libverbs
- Many solutions are “redirecting” data from normal sockets to Infiniband
 - Transparency
 - Quick and easy to use
 - Non optimal latency and throughput
- Libraries
 - FastMPJ with ibdev (MPI impl in Java)
 - MVAPICH2



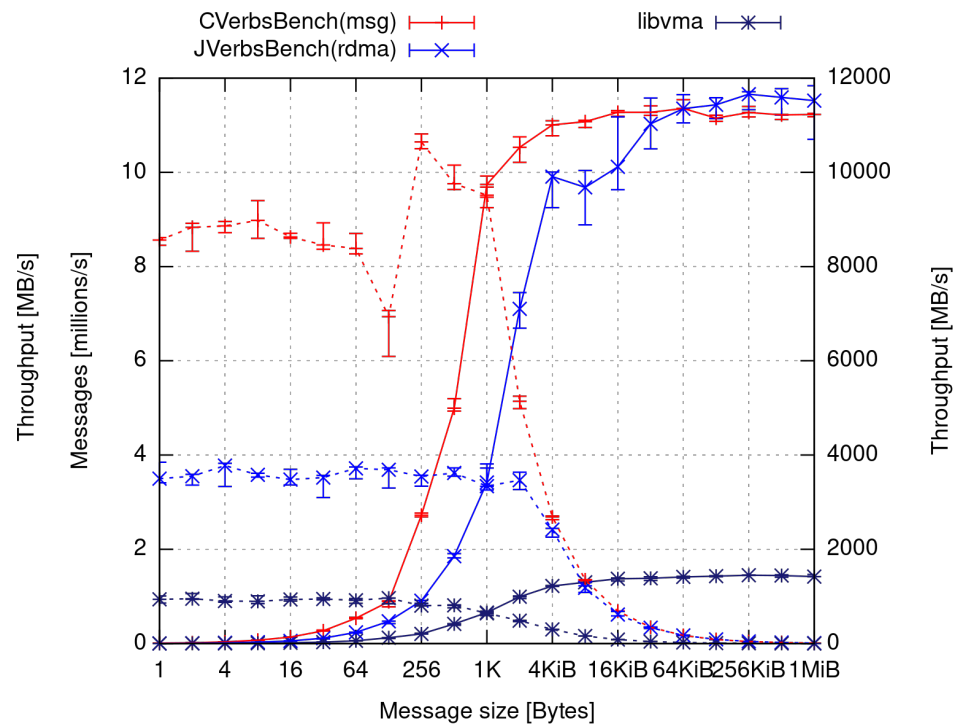
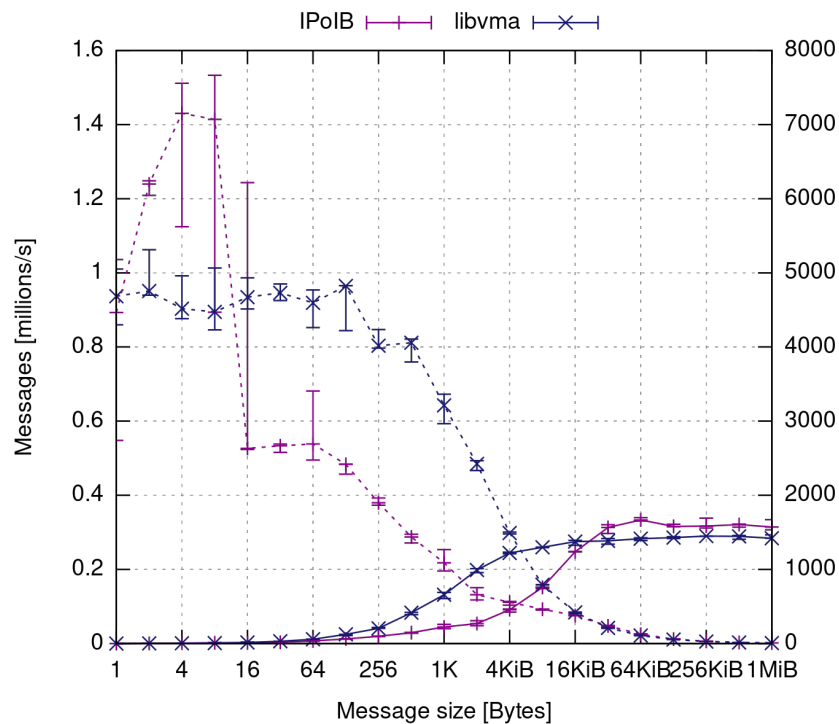
Evaluation – Sockets (uni-directional)



Evaluation – Verbs (uni-directional)



Evaluation – Sockets and Verbs (bi-directional)



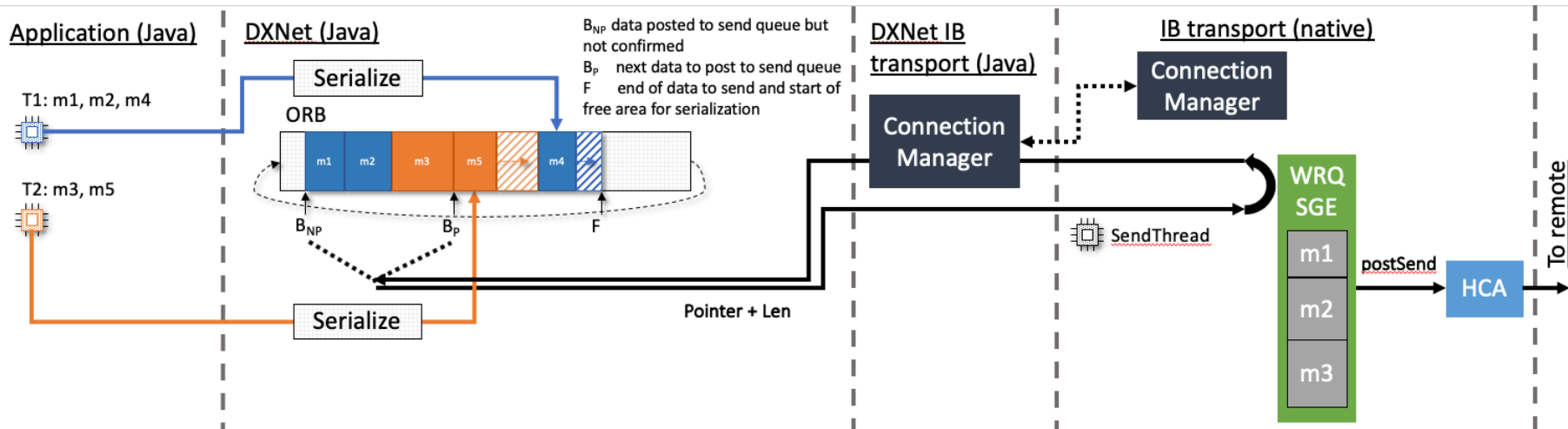


IBDXNet - Infiniband Network Subsystem

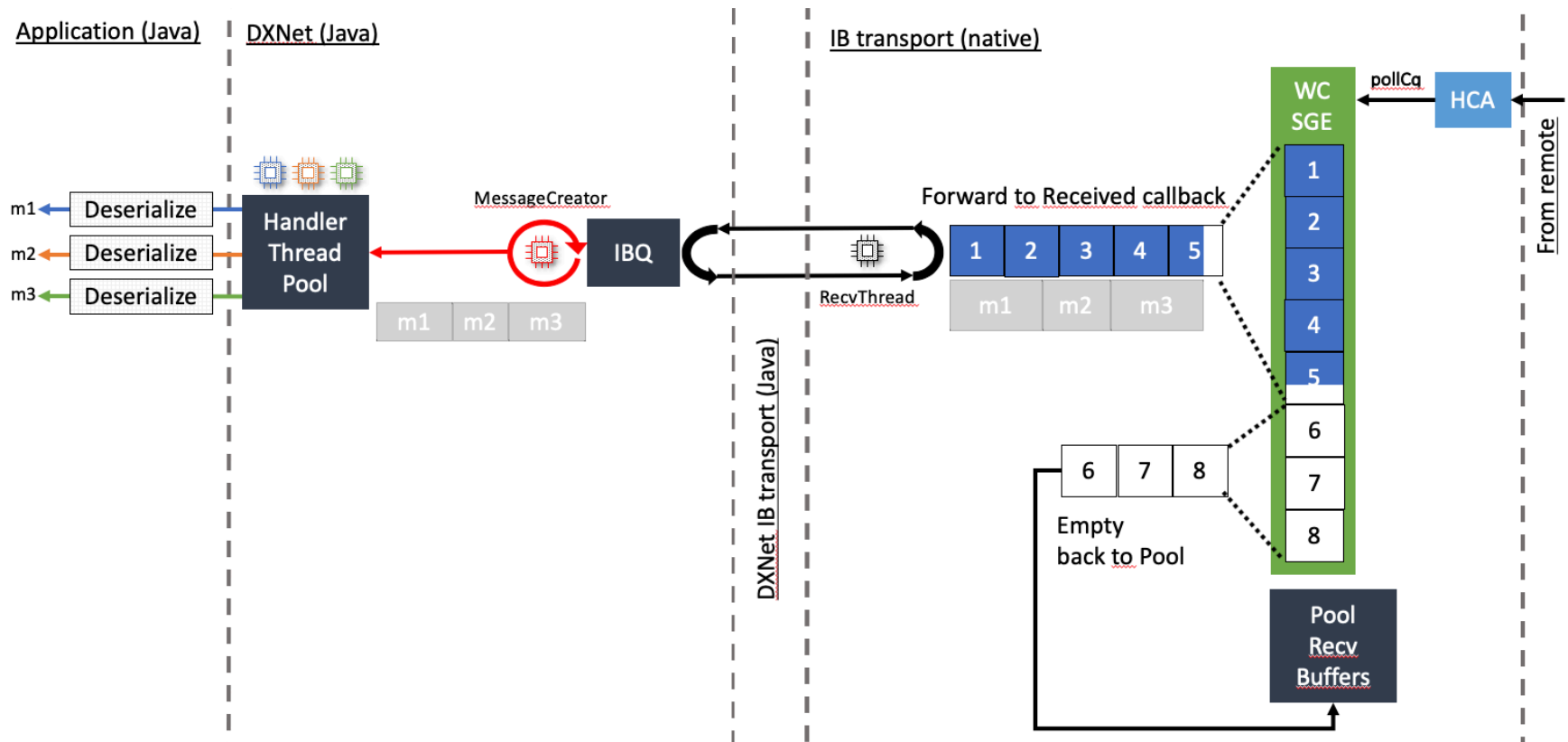
- Low latency and high throughput for Java applications as DXNet transport
- Automatic connection and queue pair management
- Asynchronous and highly optimized pipeline for sending and receiving
- Ibdxnet: Custom C/C++ native subsystem using ib-verbs
- DXNet IB transport: Java implementation of DXNet transport interface to connect to native Ibdxnet using JNI
- JNI layer to connect native IBDXNet with DXNet IB transport implementation: carefully designed and optimized for low overhead



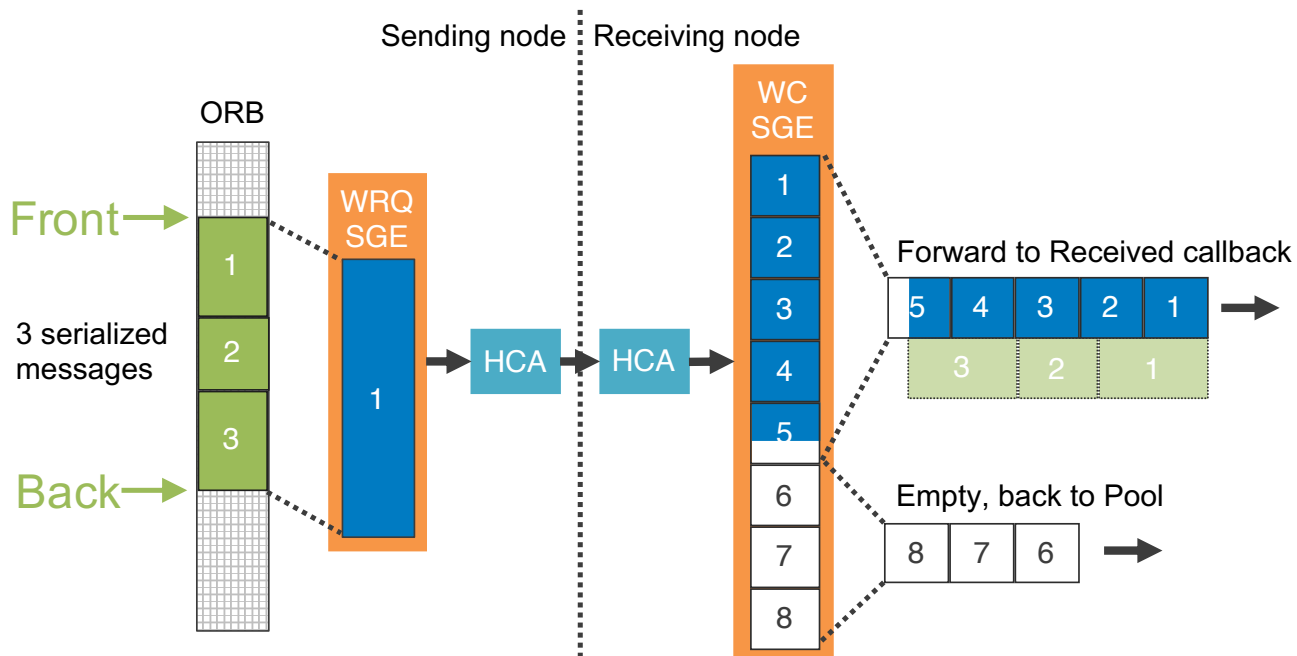
Send path and pipeline



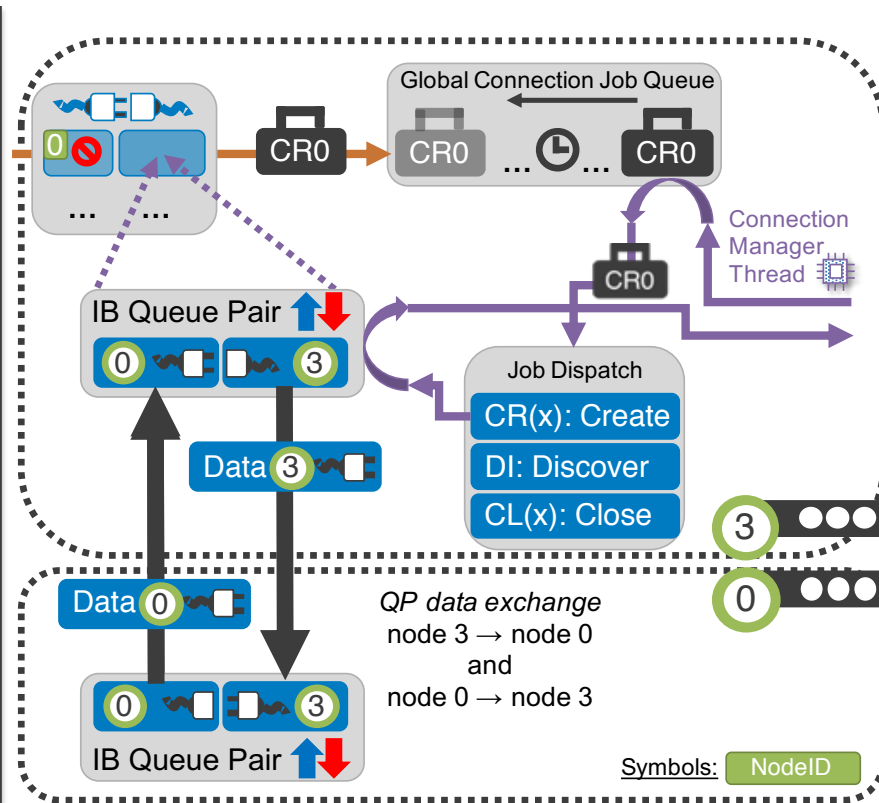
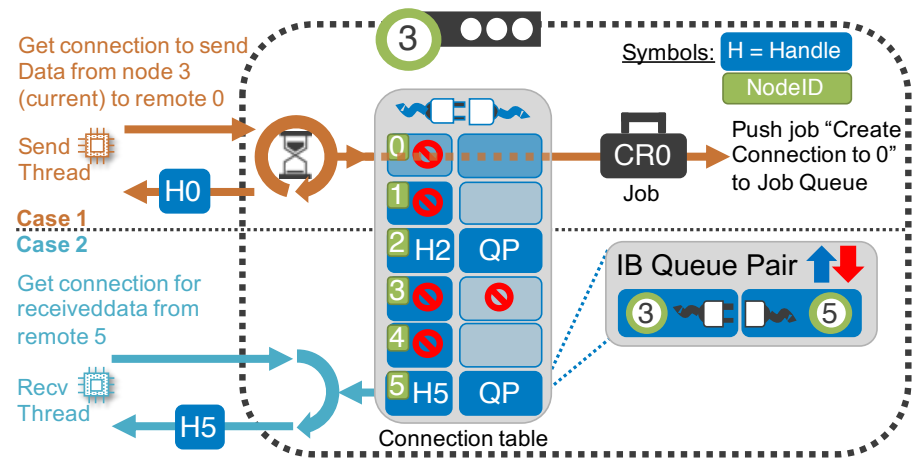
Receive path and pipeline



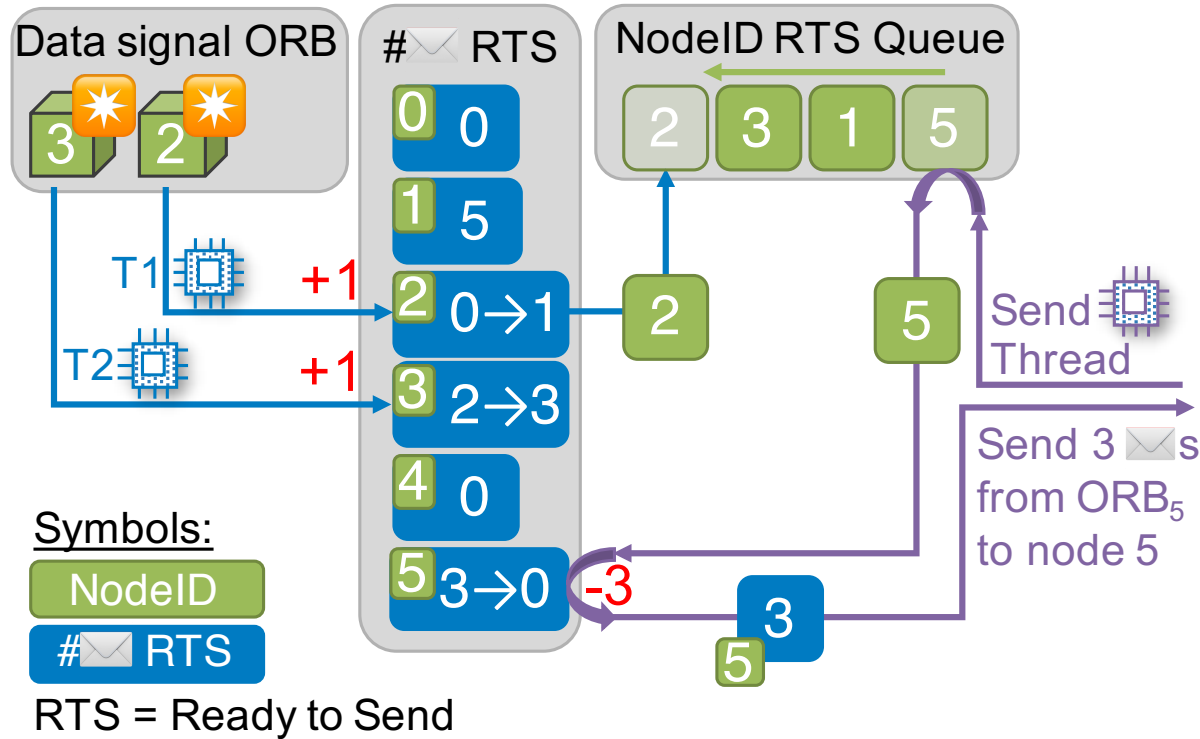
Sending and Receiving – Optimize usage of SGEs



Automatic connection management



Write Interest Manager – Data ready to send



Messaging vs. MPI

- MPI = Message Passing Interface
 - A standard used mainly in HPC for decades to solve distributed computations
- Messaging = Sending/receiving of application “messages” for means of communication with remote nodes
 - Message: A string, byte sequence, (serialized) object
- No similar messaging systems (with InfiniBand support) available in Java
- MPI can be used similar to DXNet but is still different in many aspects (see full report)
 - Even many MPI implementations do not implement (native) support for IB

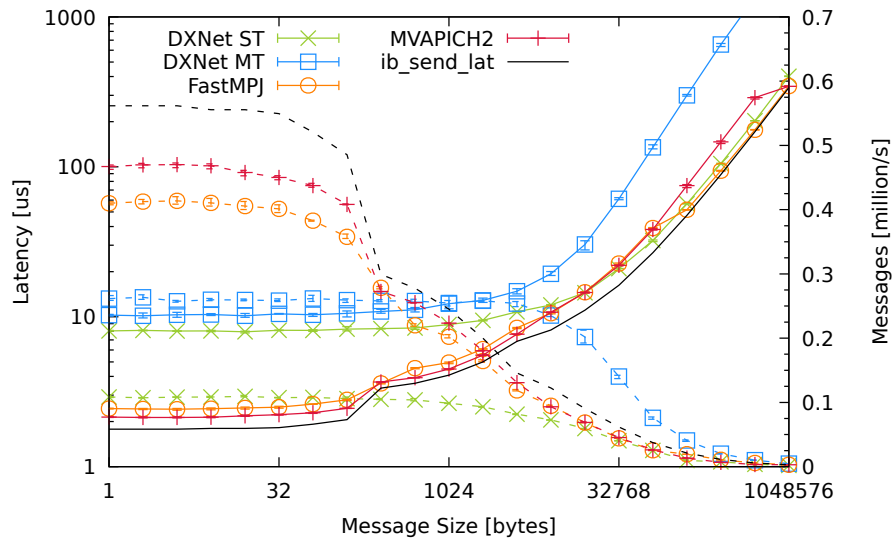
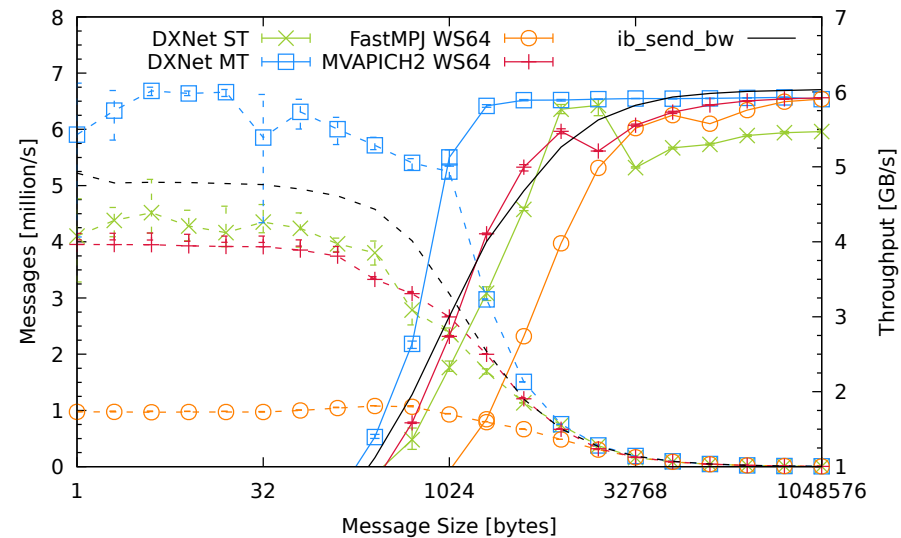


Evaluation

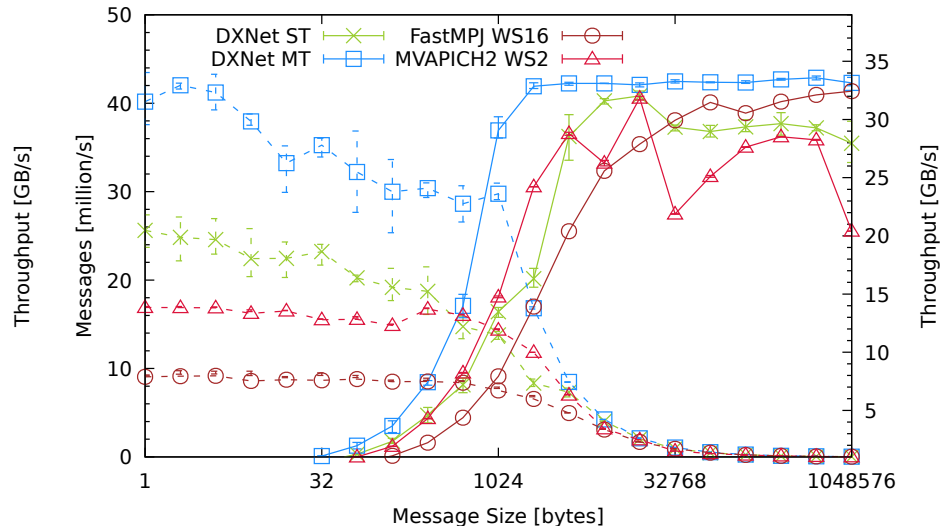
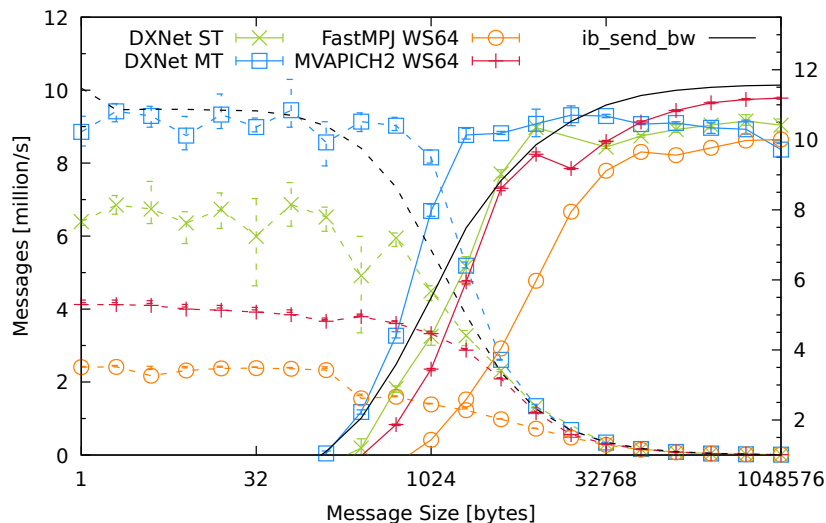
- Typical network microbenchmarks
 - Throughput: Uni- and bi-directional end-to-end
 - Ping pong latency
 - All-to-all throughput (up to 8 nodes)
- Metrics: Throughput and latency
- On 56 Gbit/s InfiniBand hardware
- Comparing to:
 - FastMPJ (MPI, Java)
 - MVAPICH2 (MPI, C)
- DXNet supports multi-threading while FastMPJ and MVAPICH2 do not
- YCSB benchmark
 - Compare DXRAM (using DXNet with IB) to RAMCloud
 - See full paper



Evaluation – Throughput and latency (uni-dir)



Evaluation – Throughput and nodes (bi-dir)



DXLog: Fast logging to SSD

- Backup Zones
- Logging Architecture
- 2-Level Logging
- Backup-side Version Control
- Evaluation (YCSB): Aerospike, Redis

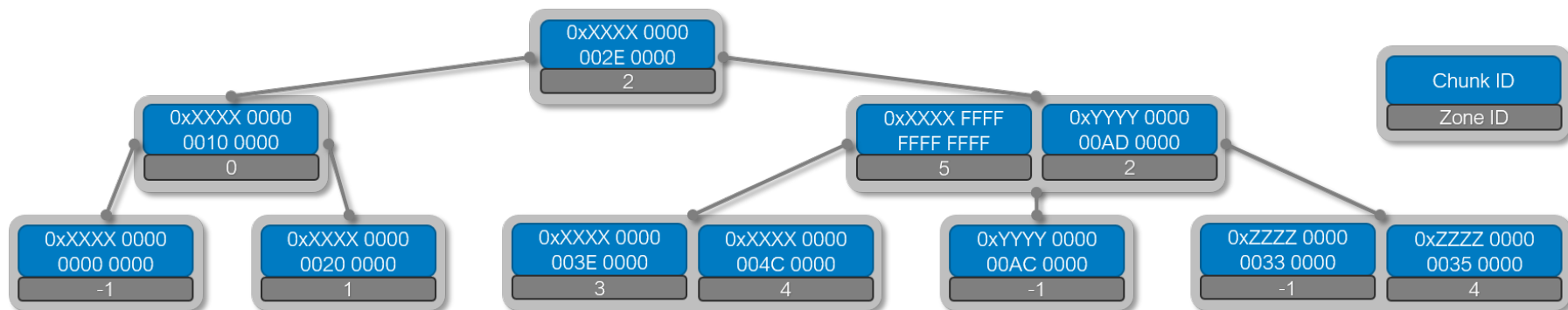
Backup Zones

- Scatter chunks of one peer to many backup peers:
 - Aggregate processing power and SSD & network bandwidth for logging and recovery.
- Each storage peer determines its own backup zones:
 - Every chunk belongs to a backup zone of a fixed size (default 256 MB).
 - Backup peers are chosen randomly, disjunctive or location-aware with configurable replication factor and fixed replication ordering.
 - Superpeers are informed on backup zone creation, only:
 - Low overhead
 - Superpeers know all backup zones of all storage peers and its backup peers
→ Important for recovery coordination.



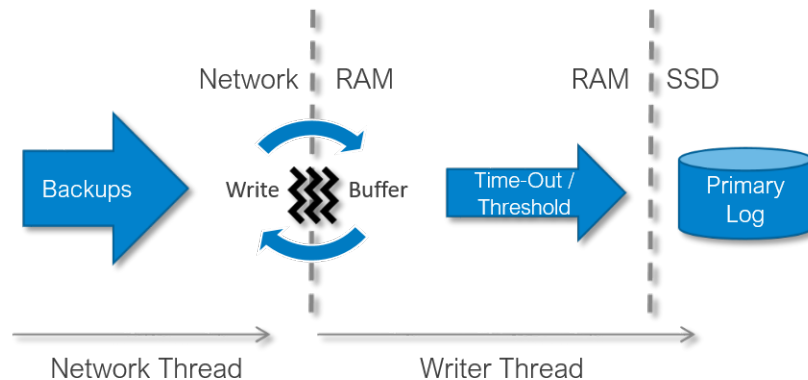
Backup Zones – Backup Zone Tree

- A backup zone might consist of locally created, immigrated and recovered chunks.
- Updating or deleting a chunk → Inform all backup peers storing replicas on.
- B-tree to store the backup zone affiliation of every chunk:
 - Stores (beginning chunk ID, end chunk ID, zone ID) tuples.
 - Overhead for a billion locally created chunks: 3 to 4 KB.



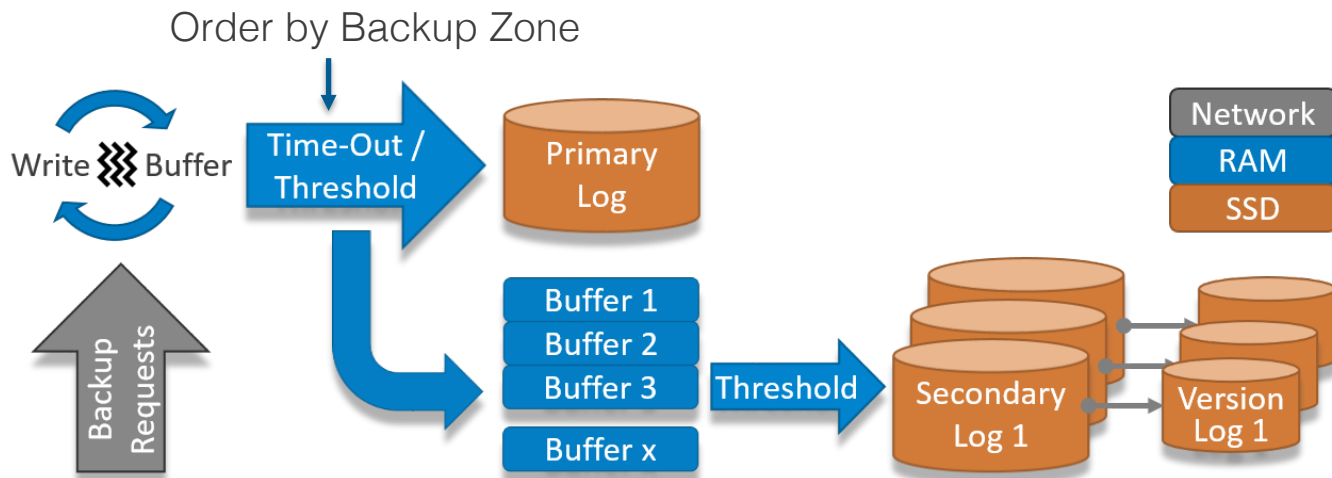
Logging Architecture

- Storing replicas on remote peers:
 - Replicating in RAM is too expensive (inefficient RAM usage).
 - Updating in-place on SSD: Too slow for small objects.
- Append data to a log:
 - Best SSD utilization
 - Low RAM consumption
 - But, requires reorganization and version control



2-Level Logging

- Primary Log: stores chunks on SSD as fast as possible.
- Secondary Logs: sort chunks by backup zone to speed up recovery.
- Version Logs: store version numbers beneath logs for recovery and reorganization.



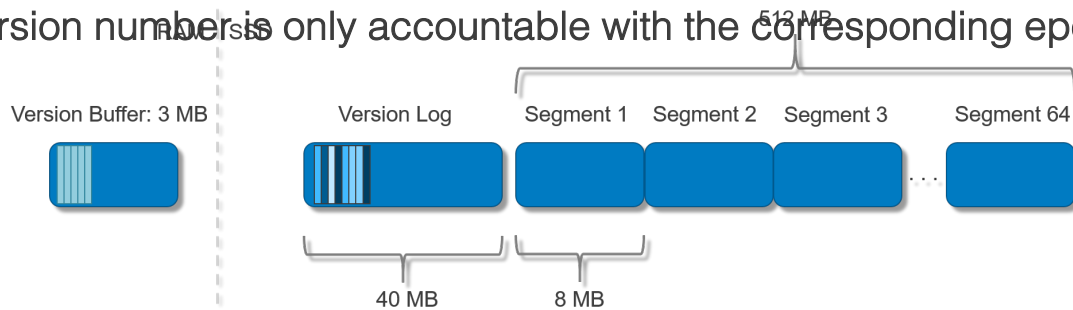
Backup-side Version Control

- Version numbers are irrelevant on storage servers but are needed for reorganization and recovery to identify valid chunks.
 - Outdated chunks have smaller version number.
 - Deleted chunks have version number -1 (no tombstones needed).
- Simple version control not applicable:
 - Versions cannot be stored in RAM (too expensive, again).
 - Simple caching is useless as access is random.
- Writing versions to SSD:
 - Reading versions from SSD before writing an object decreases performance dramatically.



Backup-side Version Control

- Solution: using epochs (combining caching and writing to SSD):
 - A fixed number of versions per secondary log is stored in buffers (called version buffers).
 - Version buffers are flushed to SSD if a threshold is reached (e.g. 3 MB).
 - After flushing, the version buffer is empty and enters a new epoch.
 - In every epoch all registered version numbers start with version 0 (no need to read current version number from SSD).
 - The version number of already registered chunks are incremented.
 - A version number is only accountable with the corresponding epoch number.



Backup-side Version Control

ID	Epoch	Version
7	1	0

#Entries: 1
Threshold: 3

[Epoch 1, Version 0] > [Epoch 0, Version 1],

because:

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

ID	Epoch	Version
7	0	1
3	0	0
4	0	0



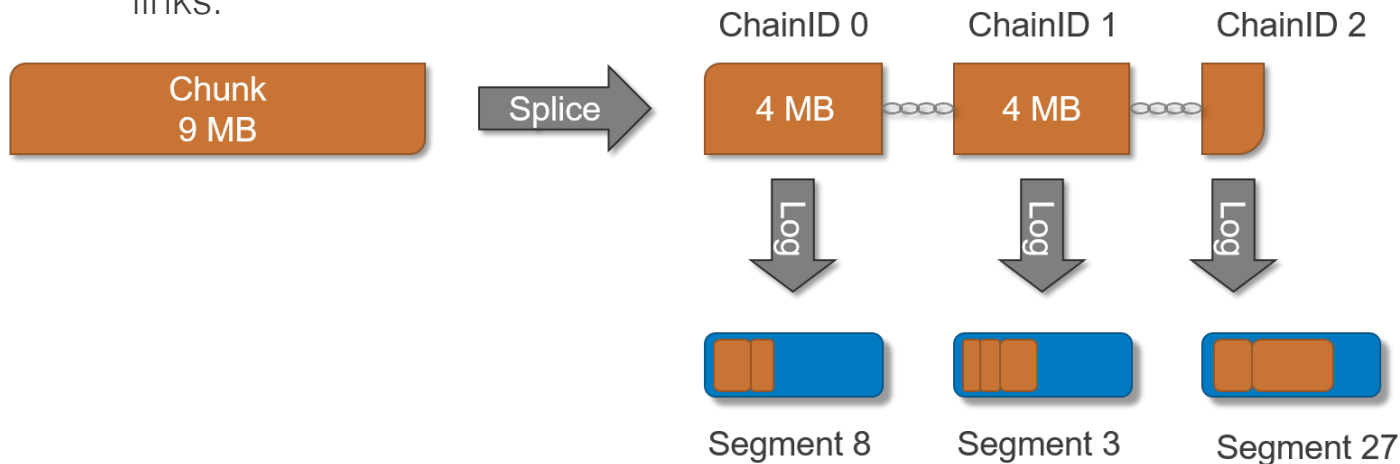
Reorganization

- Every secondary log is split into 8 MB segments.
- Reorganization steps:
 1. Read-in version log.
 2. Choose segment to reorganize by time since last reorganization.
 3. Read segment from SSD.
 4. Iterate over buffered segment and validate every log entry by comparing with current version number and verifying checksum.
 5. Move valid entries within buffered segment by overwriting invalid log entries.
 6. Write back cleansed segment.
 7. Repeat steps 2 – 6 several times (default 20).



Large Chunks

- Large chunks (e.g. > 1 MB) are seldom in target application domains but cannot be ignored.
- Example: In a social network graph, a celebrity might be connected to millions of other users → The chunks for storing all edges might exceed segment size.
- Solution: splicing and chaining.
 - Every chain link can be identified, validated and error-checked without the other chain links.





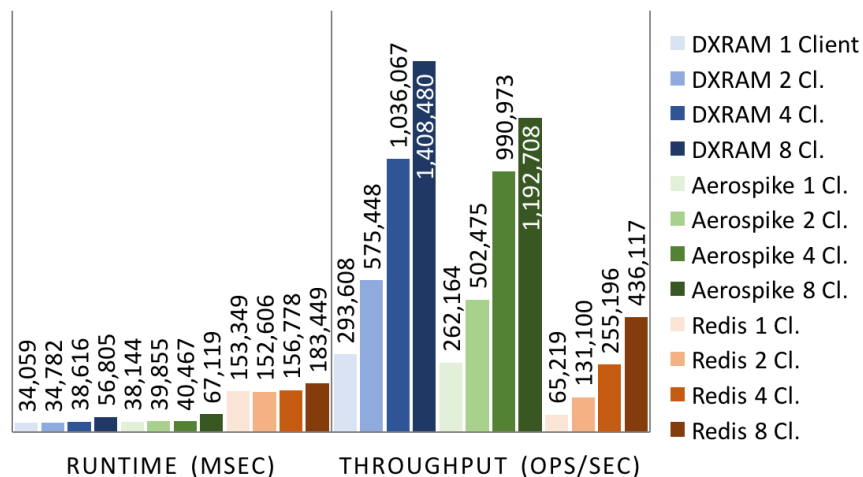
Yahoo! Cloud Serving Benchmark

- YCSB was designed to quantitatively compare distributed serving storage systems.
 - A set of simple operations (reads, writes range scans) and a tabular key-value data model.
- Workloads:
 - Workload A: 10 100-byte objects per key, 10,000,000 keys, zipfian distribution, 50% read and write operations.
 - Workload B: Identical to A, but 95 % read and 5 % write operations.
 - Workload G: One 64-byte object per key, 100,000,000 keys, zipfian distribution, 90 % read and 10 % write operations, 10,000,000 operations.

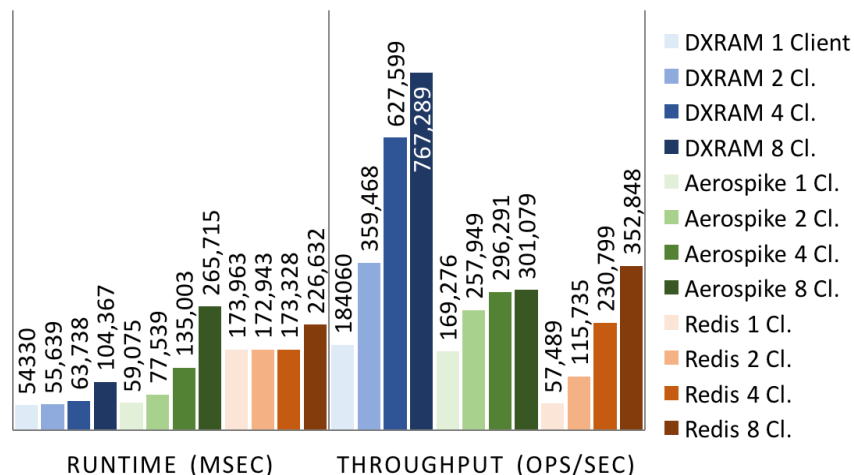


Logging Performance with YCSB!

- Setup: 16x Intel Xeon E3-1220, 16 GB RAM, 240 GB SSD, connected with Gigabit Ethernet
- We used 8 storage servers and up to 8 YCSB clients (180 threads each) for benchmarking.



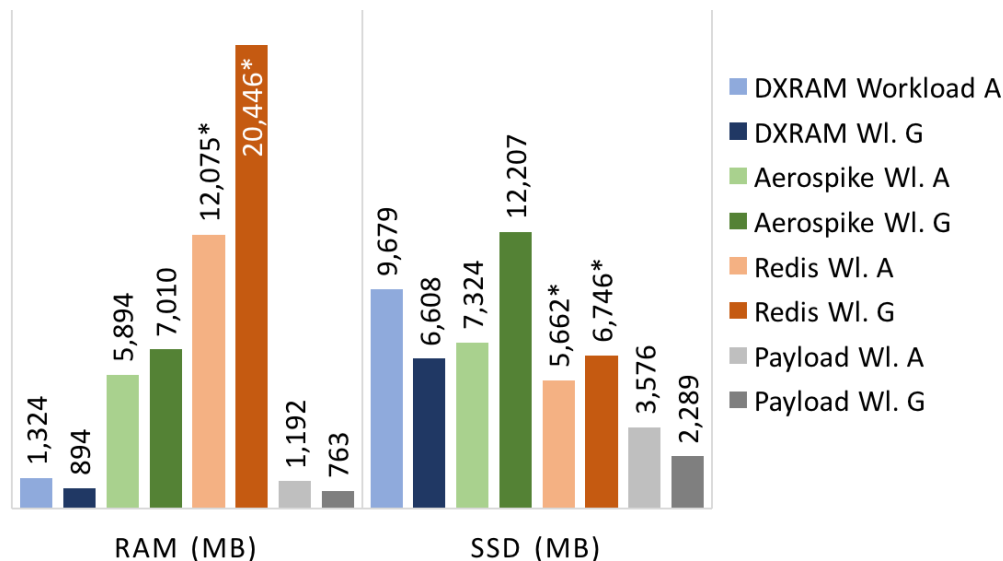
Workload G:
100,000,000 64-byte objects
90% read, 10% write



Workload A (write-heavy):
10,000,000 100-byte objects
50% read, 50% write

Logging Performance with YCSB!

- Memory consumption during YCSB! performance evaluation:

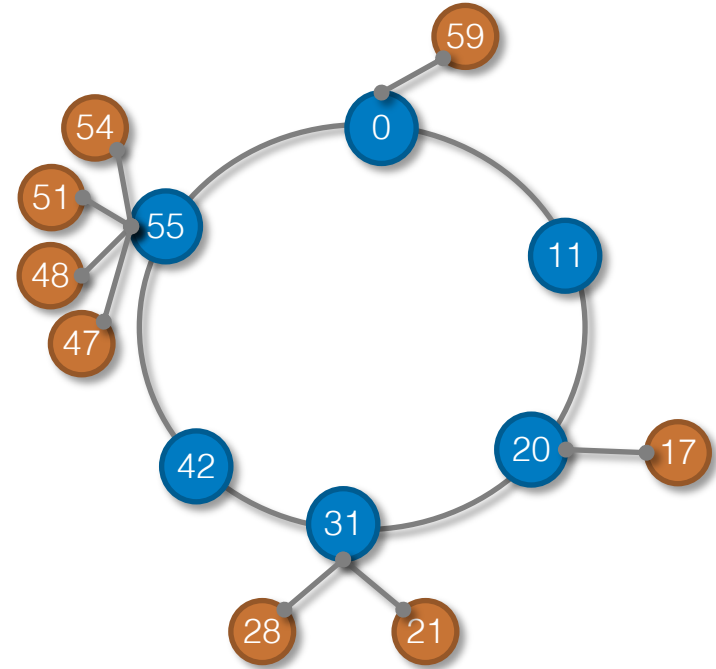


Recovery

- Failure Detection
- Recovery Initialization
- Local Recovery
- Metadata Update
- Evaluation (Azure): YCSB and RAMCloud

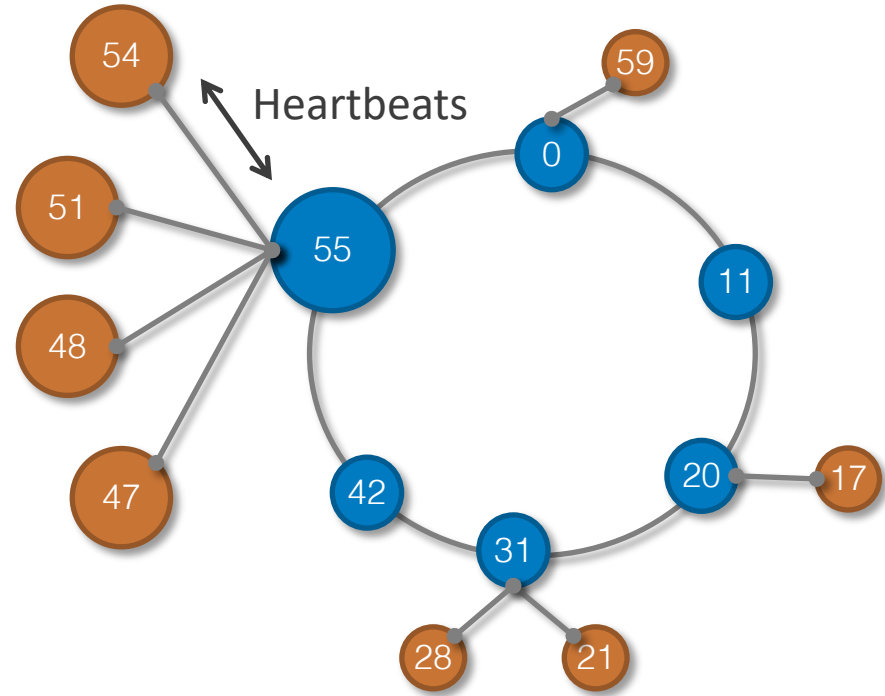
Failure Detection

- Fail-stop server failure detection:
 - Based on superpeer overlay



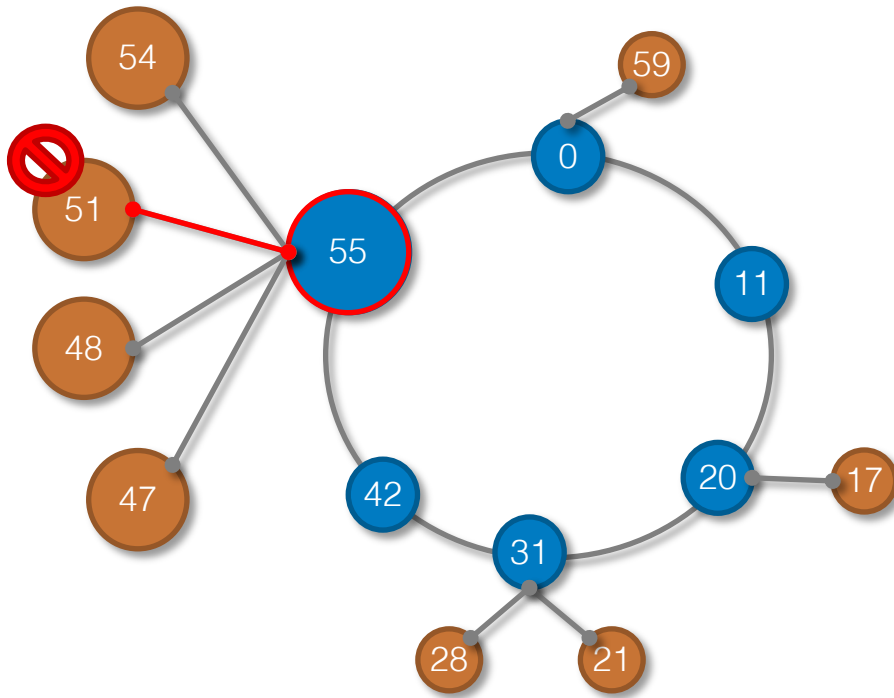
Failure Detection

- Fail-stop server failure detection:
 - Based on superpeer overlay
 - Configurable heartbeats



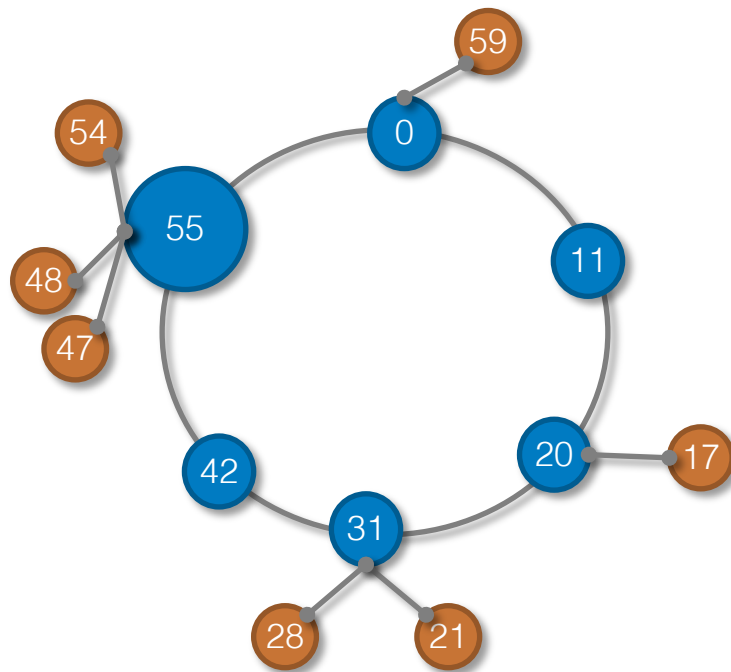
Failure Detection

- Fail-stop server failure detection:
 - Based on superpeer overlay
 - Configurable heartbeats
- Event system to handle failures.
- Peers or other superpeers might detect a failure earlier than the responsible superpeer
→ Inform responsible superpeer about failure.



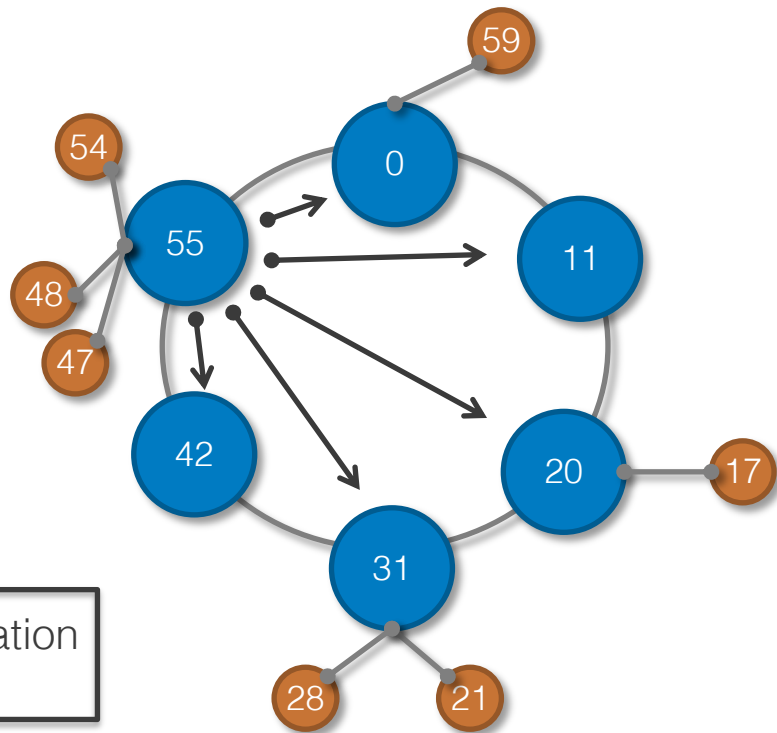
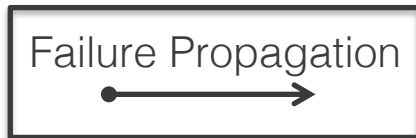
Failure Detection

- Fail-stop server failure detection:
 - Based on superpeer overlay
 - Configurable heartbeats
- Event system to handle failures.
- Peers or other superpeers might detect a failure earlier than the responsible superpeer → Inform responsible superpeer about failure.



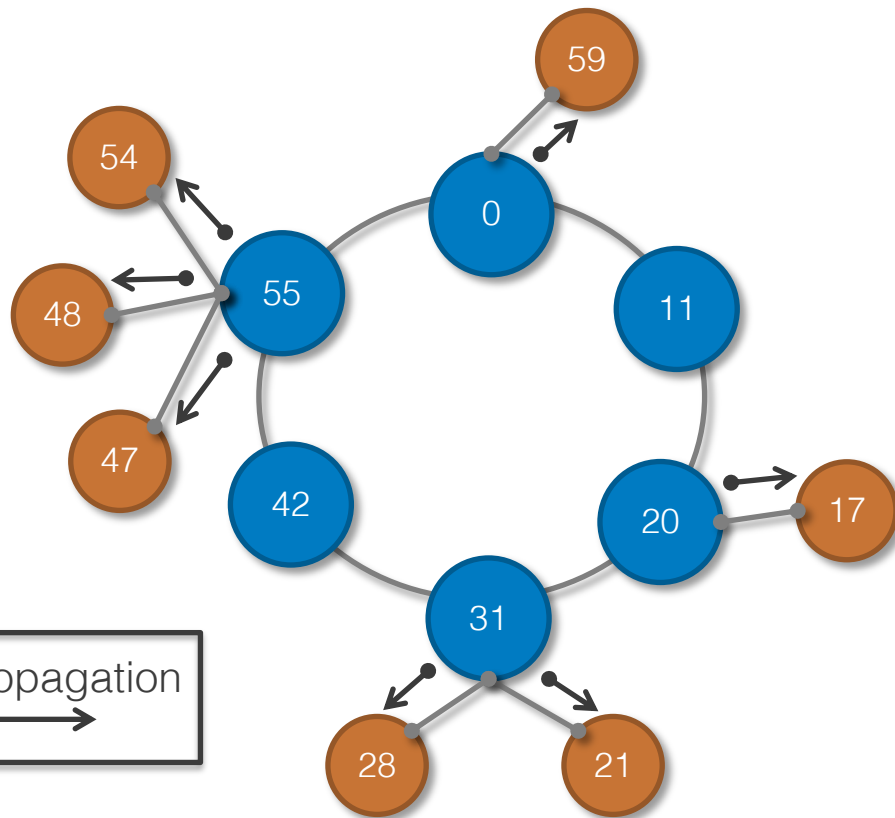
Failure Detection

- Fail-stop server failure detection:
 - Based on superpeer overlay
 - Configurable heartbeats
- Event system to handle failures.
- Peers or other superpeers might detect a failure earlier than the responsible superpeer → Inform responsible superpeer about failure.
- Superpeer propagates failure to all superpeers which inform all peers.



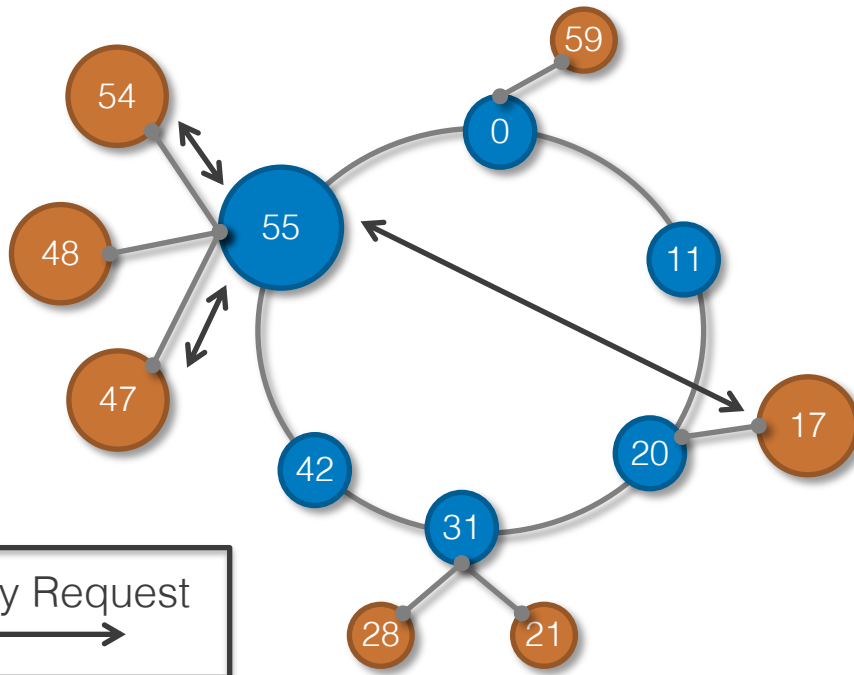
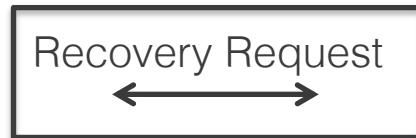
Failure Detection

- Fail-stop server failure detection:
 - Based on superpeer overlay
 - Configurable heartbeats
- Event system to handle failures.
- Peers or other superpeers might detect a failure earlier than the responsible superpeer → Inform responsible superpeer about failure.
- Superpeer propagates failure to all superpeers which inform all peers.



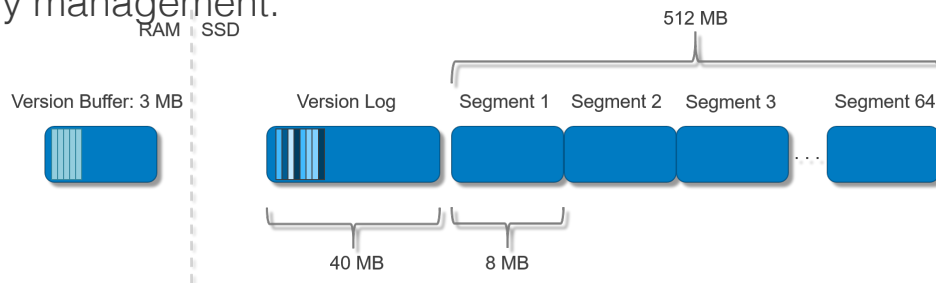
Recovery Initialization

- The backup/recovery ordering is determined on backup zone creation.
- If the first backup server is unavailable, the second will be notified and so on.
- Send all requests for all backup zones of the failed server at once for maximal parallelization.
- Collect the responses after the recovery was completely initialized.
- If response is missing, the recovery is initialized on the next backup replica of that backup zone.
- Update metadata on superpeers.
- Re-replicate chunks (once, only)

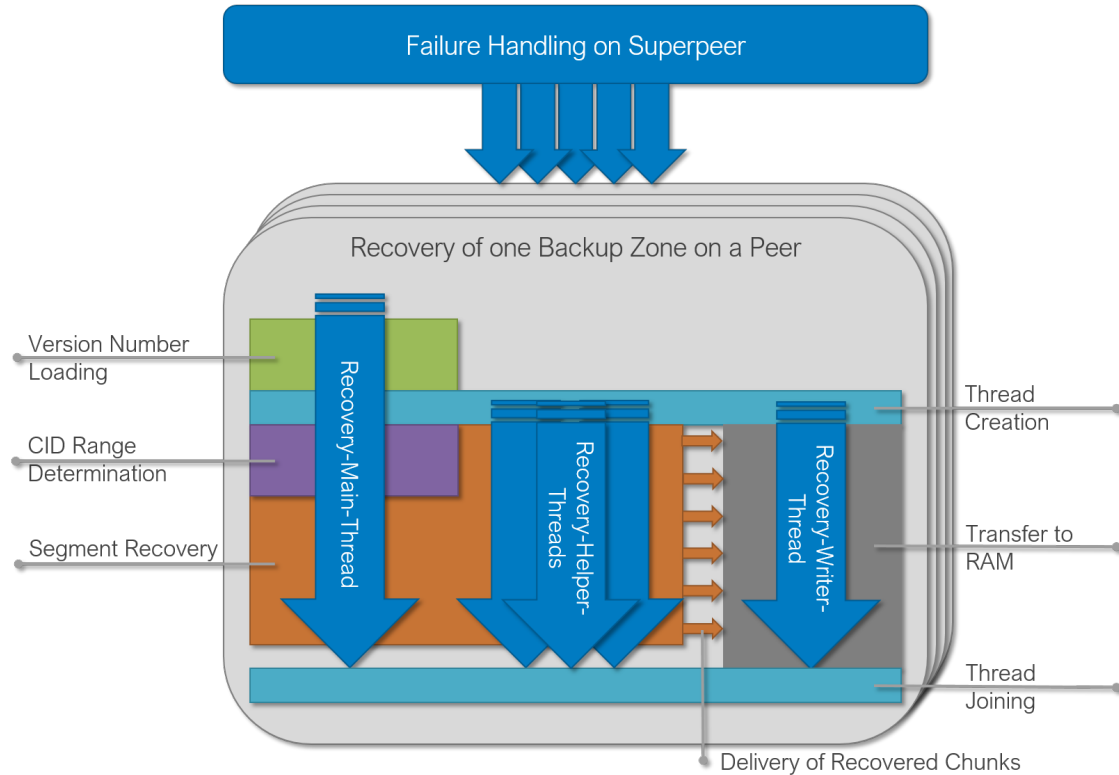


Local Recovery - Steps

1. Flush all corresponding log buffers.
2. Load version log for the secondary log from SSD for fast access.
3. Recover segment by segment as follows:
 - a) A segment is read into a byte buffer (default segment size: 8 MB).
 - b) Analyze every chunk by iterating over the byte buffer.
 - The analysis includes validation (compare version numbers) and error detection.
 - c) Store valid and error-free chunks 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.
4. Remove the secondary log from SSD.



Local Recovery - Parallelization



Microsoft Azure Testbed

- The following experiments were executed in Microsoft's Azure cloud in Germany Central with up to 72 virtual machines of type Standard_DS13_v2:
 - 8 cores (Intel Xeon E5-2673), 56 GB RAM, 112 GB SSD (max. cached throughput: 256 MBps) and 5 Gbit/s Ethernet.
- Used Microsoft Azure scale sets:
 - Set 1: Two identical scale-sets (one scale-set is limited to 40 VMs) based on a custom Ubuntu 14.04 image with 4.4.0-59 kernel.
 - Set 2: Debian 8 image and 3.16.0-4 kernel for RAMCloud (hardware and configuration identical to 1).

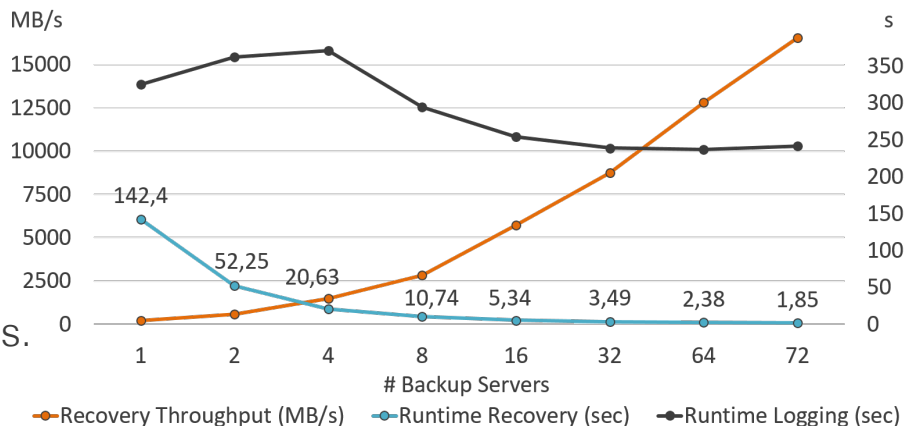


Recovery Performance

- Recovery benchmark steps:
 - The server creates 500,000,000 64-byte chunks with a total payload of more than 30 GB, allocated to 144 backup zones.
 - The data is then replicated to up to 72 slaves.
 - After logging all chunks, the master is killed which initiates the recovery process for all 144 backup zones.

- Results:

- The recovery times improve with the number of backup peers.
- With 72 backup peers the complete recovery process took less than 2 s.
- Recovery throughput >16.5 GB/s.



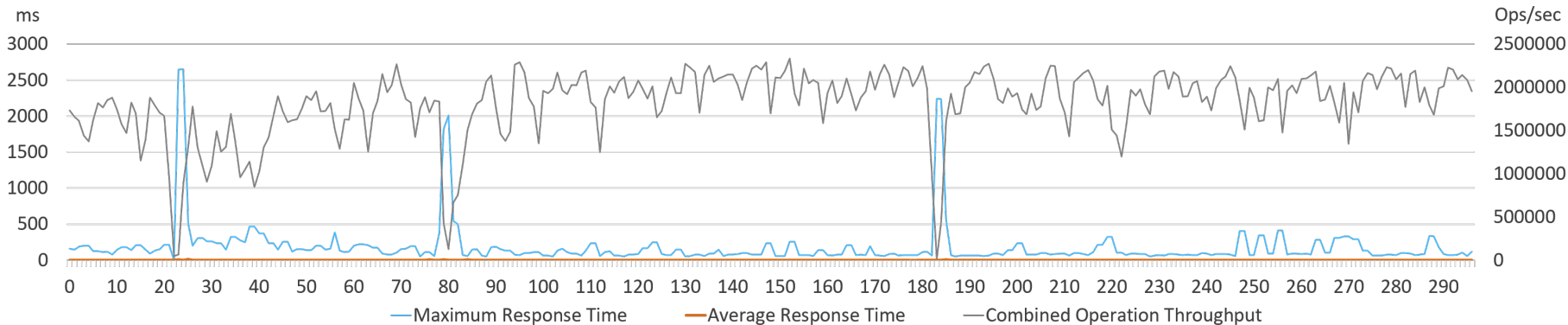


YCSB Benchmark – Recovery under High Load

- Experiment setup:
 - YCSB workload: 10 100-byte objects per key, 15,000,000 keys, zipfian distribution, 90% read and 10% write operations.
 - 48 storage servers, with a total of 7.2 billion 64-byte chunks in RAM and 21.6 billion log entries on SSD.
 - 24 YCSB clients for benchmarking. Each YCSB client is configured to emulate 100 clients using one thread per client.
- During the benchmark phase, three masters are shut down to analyze the recovery performance with high overall system load.



YCSB Benchmark – Recovery under High Load

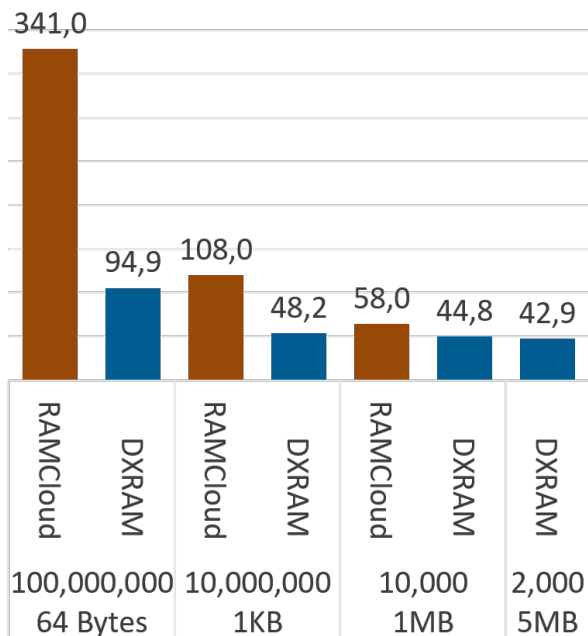


- Results: The 24 clients executed around 2 million operations per second. The maximum response time of all clients over the whole time is around 2.6 seconds, recorded during the first failure (see figure below). The average response time is around 1.3 ms.

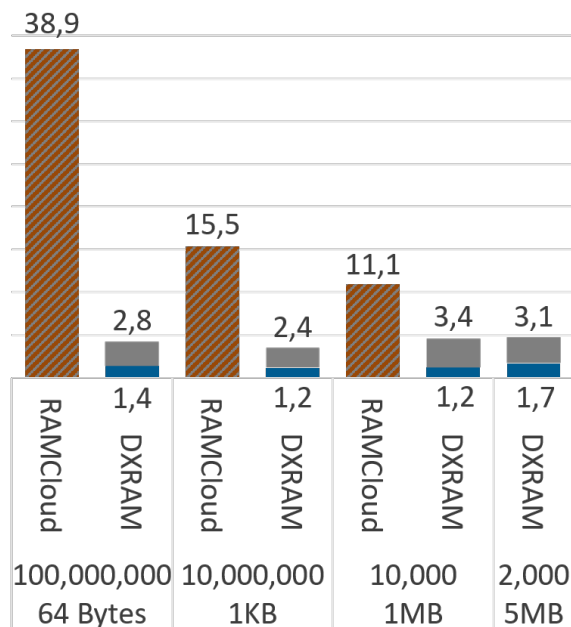
Recovery – Comparison with RAMCloud



Logging
Times (in s):



Recovery
Times (in s):





Recovery – Comparison with RAMCloud

- Why is DXRAM faster than RAMCloud in this scenario?
 - RAMCloud uses a log in RAM and distributes exact copies of the log segments to SSD on backup servers → Every object of the failed server could possibly be in every segment / on every backup server (in different versions as well).
 - 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.
 - 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.
- In RAMCloud, every object is sent over the network four times during recovery whereas in DXRAM once, only.



DXRAM: Computations

- Motivation
- Job Service
- Master Slave Service



Motivation

- Computations on Storage Nodes benefit from locally stored chunks
 - Reduce latency and network load
- Use available CPU resources





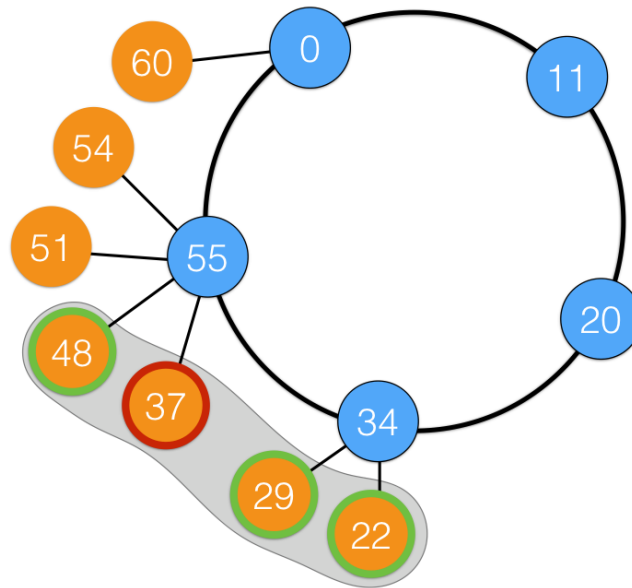
Job Service

- Deploy **light weight** jobs to single nodes.
- A job...
 - ...implements a simple interface (execute function)
 - ...runs on a **single node** on a **single core** once per deployment
 - ...has access to all DXRAM services (Network, Storage, Nameservice, Job Service, ...)
- Job Service
 - Configurable amount of **worker threads**
 - Local scheduling to workers by **work stealing**
 - Jobs can be submitted to remote nodes/Job services
 - **Load balancing** of Job Services: job queue full, hot spot, data locality



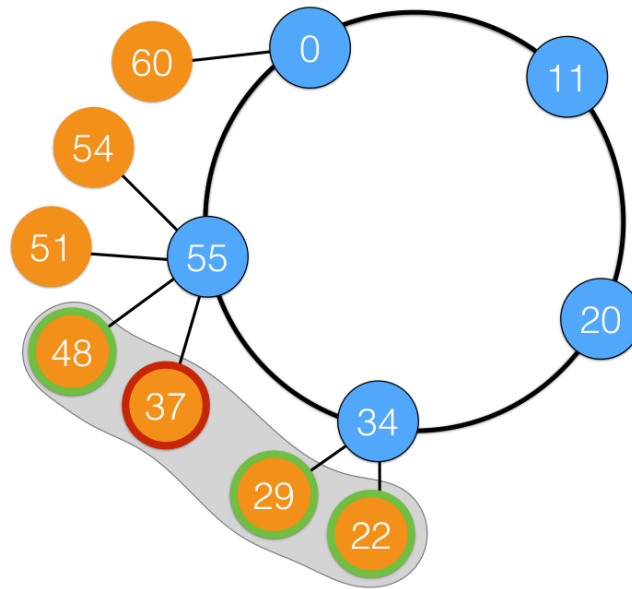
Master Slave Service

- Peers form a **compute group**
- Groups can grow
- Access to other nodes outside group (storage)
- Master**: one peer as coordinator
- Slaves**: further peers as distributed workers
- Tasks** are submitted to compute groups



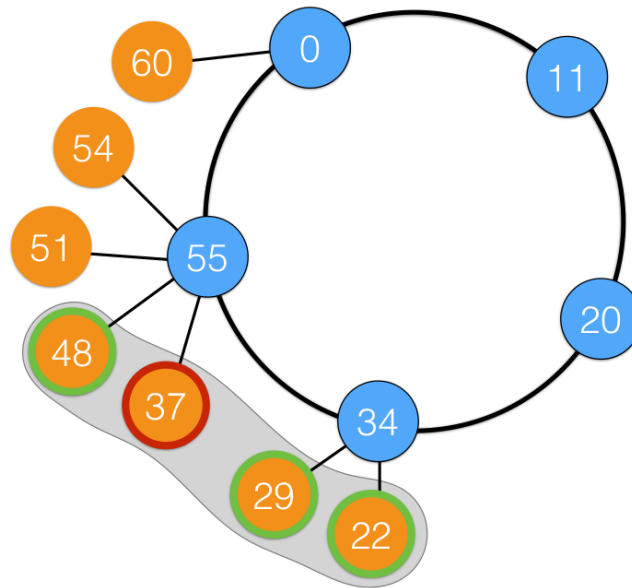
Master Slave Service - Tasks

- Managed inter node concurrency
- (Reusable) **Unit of execution**
 - Loading data
 - Processing step
 - Printing of data, statistics, results
- An implemented task runs (initially) on a **single core** on **every slave node** of the compute group concurrently
- Further forking (multiple threads) during task execution
- Superstep synchronization before and at the end of each task



Master Slave Service - Tasks

- Task context on execution
 - Compute group ID
 - Own slave ID
 - List of node IDs of every other slave
 - Total number of slaves
- Task Script
 - Chain of Tasks
 - Implicit synchronization after every task
 - Flow control: Conditions



DXGraph: Graph Processing

- Breadth-First Search



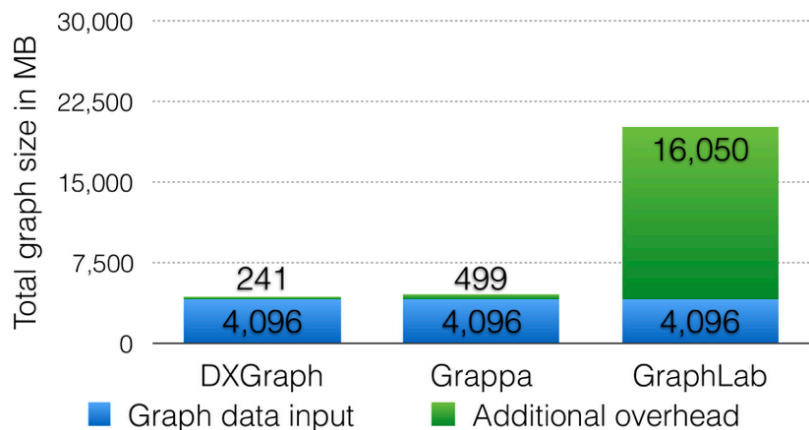
Breadth-First Search

- Implementation as specified by the Graph500 benchmark
- Stress test for system: Highly random access
- Standard top-down combined with bottom-up approach (reducing number of visited vertices)
- Compute task: Implements BFS
 - Distributed and multithreaded implementation
 - Delegates processing of non local vertices to owner node
 - Lock-free bitmap based data structure
 - Low overhead synchronization between BFS levels

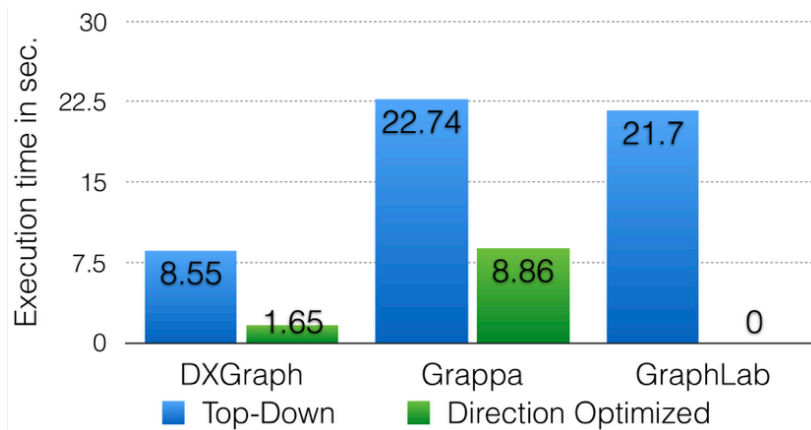


BFS: DXGraph, Grappa and GraphLab

- Scale 24 RMAT graph (Graph500 generator)
- Private cluster, 4 nodes connected by Gigabit Ethernet



Total memory consumption



Avg. BFS execution times

BFS on Hilbert

- HPC system of our university:
 - BULL: Cluster architecture, 112 nodes with 24 cores and 128 GB RAM each
- Running DXGraph's BFS implementation on BULL cluster with Gigabit Ethernet network
- Goals: Scalability, low memory overhead \Rightarrow storing many small objects
- Graph sizes tested: Scale 28 (64 GB) to 32 (1 TB)
- Random but equally distributed to 8 - 104 compute nodes



BFS on Hilbert - Results

